# Procesamiento Paralelo. Balance de Carga Dinámico en Algoritmos de Sorting

R. Marcelo Naiouf[1]

Directores: Gregory J. Randall[2]
y Armando E. De Giusti[3]

Tesis presentada para la obtención del
Grado Académico de Doctor en Ciencias

Facultad de Ciencias Exactas
Universidad Nacional de La Plata

Junio de 2004

[1]Facultad de Informática, Universidad Nacional de La Plata
[2]Universidad de la República,Uruguay
[3]Universidad Nacional de La Plata,Argentina

# Dedicatoria

A Paula, Tomás y Pilar
A mis padres
A la memoria de mis abuelos

# Resumen

Es innegable la importancia y creciente interés en el procesamiento paralelo dentro de la Ciencia de la Computación, convirtiéndose en una de las áreas que han transformado más profundamente a la disciplina. Diversas razones justifican esta hecho, como el crecimiento de la potencia de cómputo, la existencia de problemas donde el tiempo de resolución secuencial es inaceptable, la posibilidad de mapear la concurrencia implícita del problema a procesos paralelos para minimizar el tiempo de respuesta, etc.

Un *sistema paralelo* es la combinación de un algoritmo paralelo y la máquina sobre la cual éste se ejecuta, y ambos factores poseen numerosas variantes. Respecto de los algoritmos paralelos, pueden ser especificados utilizando una diversidad de modelos y paradigmas. Por el lado de las arquitecturas de soporte, si bien todas poseen más de un procesador, pueden diferir en varias dimensiones tales como el mecanismo de control, la organización del espacio de direcciones, la granularidad de los procesadores y la red de interconexión.

Entre los objetivos del paralelismo se encuentran reducir el tiempo de ejecución y hacer uso eficiente de los recursos de cómputo. El uso desigual de los elementos de procesamiento puede causar pobre eficiencia o hacer que el tiempo paralelo sea mayor que el secuencial. El balance de carga consiste en, dado un conjunto de tareas que comprenden un algoritmo complejo y un conjunto de computadoras donde ejecutarlas, encontrar el mapeo de tareas a computadoras que resulte en que cada una tenga aproximadamente igual trabajo. Un mapeo que balancea la carga de los procesadores incrementa la eficiencia global y reduce el tiempo de ejecución.

El problema de asignación o mapeo es *NP*-completo para un sistema general con $n$ procesadores, y por lo tanto la tarea de encontrar una asignación de costo mínimo es computacionalmente intratable salvo para sistemas muy chicos. Por esta razón pueden utilizarse enfoques alternativos como la relajación, el desarrollo de soluciones para casos particulares, la optimización enumerativa o la optimización aproximada (uso de heurísticas que brindan soluciones subóptimas aunque aceptables).

En algunos casos el tiempo de cómputo asociado con una tarea puede determinarse *a priori*. En tales circunstancias, se puede realizar el mapeo antes de comenzar la computación (balance de carga *estático*). Para una clase importante y creciente de aplicaciones, la carga de trabajo para una tarea particular puede modificarse en el curso de una computación y no puede estimarse de antemano; en estos casos el mapeo debe cambiar durante el cómputo (balance de carga *dinámico*), realizando etapas de balanceo durante

la ejecución de la aplicación.

En general, el balance estático es de menor complejidad que el dinámico, pero también menos versátil y escalable. Conceptualmente los métodos dinámicos requieren alguna forma de mantener una visión global del sistema y algún mecanismo de negociación para la migración de procesos y/o datos. Si bien potencialmente pueden mejorar la performance global de la aplicación redistribuyendo la carga entre los elementos de procesamiento, esta actividad se realiza a expensas de computación útil, produce un overhead de comunicación y requiere espacio extra para mantener la información. No puede establecerse un método efectivo y eficiente en todos los casos. Siempre la elección depende de la aplicación y la plataforma de soporte, y en muchos casos es necesario adaptar o combinar metodos existentes para lograr buena performance.

El *sorting* (ordenación) es una de las operaciones más comunes realizadas en una computadora. La tarea del sorting se define como el acomodamiento de un conjunto desordenado de elementos en orden creciente (o decreciente). Numerosas aplicaciones requieren que los datos se encuentren ordenados para poder accederlos de manera más eficiente. El sorting es importante dentro del cómputo paralelo por su relación cercana con el ruteo de datos entre procesadores, parte esencial de algunos algoritmos. Muchos problemas de *routing* pueden resolverse ordenando los paquetes en sus direcciones de destino, mientras varios algoritmos de sorting se basan en esquemas de ruteo para su implementación eficiente. También la operación de ordenación es utilizada con frecuencia en el procesamiento de Bases de Datos, por ejemplo en operaciones con cláusulas *Distinct*, *Order By* y *Group By* en SQL.

Los algoritmos de sorting pueden categorizarse como *basados en comparación* y *no basados en comparación*. Los primeros ordenan comparando repetidamente pares de elementos e intercambiándolos si es necesario. Para $n$ items, el sorting secuencial basado en comparación tiene una cota inferior en tiempo de $\Theta(n \log n)$. En particular, *merge-sort* es $O(n \log n)$, lo que no permite mejoras sustanciales en los algoritmos secuenciales. Los métodos que no se basan en comparación usan ciertas propiedades conocidas de los elementos, y la cota inferior es $\Theta(n)$.

Existen numerosos algoritmos de ordenación tanto secuenciales como paralelos. El sorting paralelo incluye tanto las versiones distribuidas de algoritmos secuenciales clásicos como métodos directamente paralelos. El proceso de paralelizar un algoritmo de sorting secuencial involucra distribuir los elementos en los procesadores disponibles, lo que implica tratar temas tales como dónde se almacenan las secuencias de entrada y salida, o cómo se realizan las comparaciones. Los problemas de sorting con gran volumen de datos por procesador son los más interesantes y sobre los cuales las máquinas paralelas actuales funcionan mejor, debido a su potencia de cómputo y capacidad de memoria.

Una gran parte de los métodos de balance de carga se refieren a problemas en los que se cuenta con alguna manera de conocer cuál es la carga, por ejemplo, expresándola como una relación de la cantidad de puntos de entrada de una grilla a procesar. Resulta interesante atacar el problema del balance de la carga en aplicaciones donde el trabajo no depende del tamaño de los datos de entrada sino de determinada característica de los

mismos. Muchos de los algoritmos de sorting se encuentran dentro de esta clase.

Algunas técnicas de sorting intentan balancear la carga mediante un muestreo inicial de los datos a ordenar y una distribución de los mismos de acuerdo a *pivots*. Otras redistribuyen listas parcialmente ordenadas de modo que cada procesador almacene un número aproximadamente igual de claves, y todos tomen parte del proceso de merge durante la ejecución. Esta Tesis presenta un nuevo método que balancea dinámicamente la carga basado en un enfoque diferente, buscando realizar una distribución del trabajo utilizando un *estimador* que permita *predecir* la carga de trabajo pendiente.

El método propuesto es una variante de *Sorting by Merging* Paralelo, esto es, una técnica basada en comparación. Las ordenaciones en los bloques se realizan mediante el método de Burbuja o *Bubble Sort* con centinela. En este caso, el trabajo a realizar (en términos de comparaciones e intercambios) se encuentra afectada por el *grado de desorden* de los datos. Se estudió la evolución de la cantidad de trabajo en cada iteración del algoritmo para diferentes tipos de secuencias de entrada ($n$ datos con valores de 1 a $n$ sin repetición, datos al azar con distribución normal), observándose que el trabajo disminuye en cada iteración. Esto se utilizó para obtener una estimación del trabajo restante esperado a partir de una iteración determinada, y basarse en el mismo para corregir la distribución de la carga.

Con esta idea, el método no distribuye inicialmente entre las tareas todos los datos a ordenar, sino que se reserva un porcentaje de los mismos. Luego de una determinada cantidad de "vueltas" (en particular, con el 5% de las iteraciones), estima el trabajo restante de cada tarea basado en lo ya realizado, y distribuye dinámicamente los datos reservados de manera inversamente proporcional al trabajo restante estimado para cada tarea.

El esquema presentado utiliza el paradigma *master/slave*, y fue implementado en una arquitectura paralela con comunicación por bus (cluster de PCs homogéneas), aunque puede ejecutarse también en máquinas de memoria compartida. Las características principales del método son su sencilllez, efectividad, comunicación limitada, y posibilidad de aplicación a diferentes problemas. Se presentan resultados que muestran la bondad de las estimaciones y que se balancea la carga de trabajo adecuadamente en un alto porcentaje de los casos.

# Prefacio

El tema de esta Tesis se enmarca en el proyecto de Procesamiento Concurrente y Paralelo que el autor codirige dentro del Instituto de Investigación en Informática LIDI (III-LIDI). En particular, los aspectos de métricas del paralelismo y balance de carga son de interés en el contexto del proyecto mencionado.

El trabajo está organizado en tres Partes, y cada Parte se encuentra dividida en Capítulos.

La Parte I se refiere a aspectos teóricos generales. En el Capítulo 1 se presentan conceptos básicos de paralelismo, modelos de concurrencia y de computación paralela. En el Capítulo 2 se incluyen clasificaciones de arquitecturas paralelas por distintos criterios y se describen algunas máquinas reales. En el Capítulo 3 se presentan los tipos de paralelismo, los principales paradigmas de computación paralela y una metodología de diseño de algoritmos paralelos. En el Capítulo 4 se introduce el tema de las métricas del paralelismo, presentando distintas medidas para evaluar un sistema paralelo así como los modelos de speedup más conocidos y el análisis de escalabilidad.

La Parte II trata en el Capítulo 5 la asignación de tareas a procesadores y el balance de carga, presentando técnicas de balanceo estáticas y dinámicas. En el Capítulo 6 se analiza el tema del Sorting en general, incluyendo algunos de los métodos secuenciales y paralelos más conocidos. En el Capítulo 7 se presenta el método de sorting paralelo con balance dinámico de carga propuesto. En el Capítulo 8 se analizan los resultados obtenidos, y en el Capítulo 9 se incluyen las conclusiones y las posibilidades de trabajo futuro.

La parte III est compuesta por los apéndices. El Apéndice A se refiere a Complejidad de Funciones y Análisis de Orden. El Apéndice B describe el modelo de arquitectura y software utilizado.

Finalmente, se incluyen una Lista de Figuras y la Bibliografía.

# Agradecimientos

A mi Familia, que me apoya y brinda todo su cariño.
A Tito, quien en 1988 me dio la oportunidad de incorporarme al viejo LIDI e iniciarme en la investigación. Desde ese momento guió mi carrera como Director pero fundamentalmente como un amigo, insistiéndome para que concluya esta Tesis.
A Patricia, por su amistad y apoyo en momentos de duda.
A Laury y Franco, quienes participaron activamente discutiendo ideas y realizando experimentación.
Al Flaco Bria, quien colaboró en el análisis estadístico de la estimación.
A los miembros del III-LIDI que de alguna manera han hecho posible la concreción de este proyecto.

# Índice General

# Parte I

# Aspectos teóricos

# Capítulo 1

# Procesamiento Paralelo

## 1.1 Introducción

En la actualidad es innegable la importancia y el creciente interés en el procesamiento paralelo dentro del espectro de la Ciencia de la Computación. Desde la invención de las computadoras seriales convencionales, su velocidad se ha incrementado para cumplir con las necesidades de las aplicaciones emergentes. Sin embargo, la limitación física fundamental impuesta por la velocidad de la luz hace imposible obtener mejoras indefinidamente, y una manera natural de evitar esta saturación es usar un ensamble de procesadores para resolver problemas [209, 139]. En este sentido, el paralelismo es un concepto intuitivo.

Entre todas las ideas esparcidas por la Ciencia de la Computación en los últimos años, pocas han transformado el área de manera tan profunda como la computación paralela. Virtualmente todos los aspectos se vieron afectados, y se generó un gran número de conceptos nuevos. Desde la Arquitectura de Computadoras hasta los Sistemas Operativos, desde los Lenguajes de Programación y Compiladores hasta Bases de Datos e Inteligencia Artificial, y desde la Computación Numérica hasta la Combinatoria, cada rama sufrió un renacimiento [7, 158].

Existen diversas razones para justificar esta importancia tomada por el paralelismo, entre las que se pueden citar:

- El crecimiento de la potencia de cómputo, dado en la evolución de la tecnología de los componentes y en las arquitecturas de procesamiento (supercomputadoras, hipercubos de procesadores homogéneos, grandes redes de procesadores no-homogéneos, procesadores de imágenes, de audio, etc.)
- La existencia de problemas computacionales en los cuales el tiempo para obtener una respuesta utilizando una computadora secuencial es inaceptable.

- La transformación y creación de algoritmos que explotan la concurrencia implícita en el problema a resolver, de modo de distribuir el procesamiento minimizando el tiempo total de respuesta. Naturalmente esta transformación también debe adaptarse a la arquitectura física de soporte.

- La capacidad del cómputo distribuido/paralelo de reducir el tiempo de procesamiento en problemas de cálculo intensivo (simulaciones, búsquedas, cómputo científico) o de grandes volúmenes de información (bases de datos, imágenes, etc.).

- La necesidad de tratar con sistemas de tiempo real distribuidos, con requerimientos críticos en términos de tiempo de respuesta.

- La existencia de sistemas en los que no es tan importante la velocidad de cómputo sino la necesidad de resolver problemas en más de una ubicación física a la vez, capacidad que puede asociarse rápidamente a una configuración paralela.

- Las posibilidades que el paradigma paralelo ofrece en términos de investigación de técnicas para el análisis, diseño y evaluación de algoritmos. Conceptualmente, usar varios procesadores que trabajan juntos en una computación dada representa un paradigma interesante. Brinda ideas teóricas renovadas en la mayor parte de los problemas computacionales, independientemente de su origen o complejidad.

En teoría, el paralelismo es simple: aplicar múltiples CPUs a un único problema. Para el científico computacional resuelve algunas de las restricciones impuestas por las computadoras de un solo procesador [269]. Además de ofrecer soluciones más rápidas, las aplicaciones paralelizadas pueden resolver problemas más grandes y complejos cuyos datos de entrada o resultados intermedios exceden la capacidad de memoria de una CPU; las simulaciones pueden ser corridas con mayor resolución; los fenómenos físicos pueden ser modelizados de manera más realista.

En la práctica, el paralelismo tiene un alto precio. La programación paralela involucra una curva creciente de aprendizaje y es de esfuerzo intensivo: el programador debe pensar sobre la aplicación de maneras novedosas y puede terminar reescribiendo todo el código serial. Más aún: las técnicas para *debugging* y *tuning* de performance de programas secuenciales no se extienden fácilmente al mundo paralelo.

Cómo saber si hacer o no la inversión? El propósito y la naturaleza de la aplicación son los indicadores más importantes de cuan exitosa será la paralelización. La elección de la computadora paralela y el plan de ataque tendrán impacto significativo no sólo en la performance sino también en el nivel de esfuerzo requerido para lograrla.

Entre los componentes necesarios para la computación paralela se encuentran la arquitectura (hardware) y el sistema operativo, lenguaje y compilador (software). Sin embargo, lo más importante son los *algoritmos paralelos* (o métodos de solución paralelos). De la

misma manera en que los algoritmos ocupan un lugar central en la Ciencia de la Computación, los algoritmos paralelos son el corazón de la computación paralela, junto con los métodos utilizados para su construcción y análisis de performance.

Es importante referirse a un algoritmo paralelo mencionando el *modelo de computación paralela* para el que se lo diseñó. Esto se debe a que, a diferencia de la computación secuencial, donde la mayoría de las máquinas pertenecen a un mismo modelo, se han propuesto y usado un gran número de modelos para estudiar la computación paralela en teoría y para construir máquinas paralelas en la práctica. Estos modelos difieren de acuerdo a si los procesadores se comunican entre sí por memoria compartida o por una red, si la interconexión es en forma de arreglo, árbol o hipercubo, si los procesadores ejecutan el mismo o distintos algoritmos, si los procesadores operan sincrónica o asincrónicamente, etc. Ninguno de los modelos ha logrado imponerse definitivamente, ya que cada uno enfatiza determinados aspectos a costa de otros. Una razón más sutil es que no es probable una máquina paralela universal; para cada aplicación existe una máquina óptima.

Para ilustrar algunos aspectos importantes del cómputo paralelo se puede trazar una analogía con un escenario real, como es el problema de apilar en sus estantes un conjunto de libros en una biblioteca. Un único trabajador no podría realizarlo más rápido que una determinada velocidad, pero se puede acelerar el proceso empleando más de un trabajador. Si se asume que los libros están organizados en estantes y los estantes se agrupan en compartimentos, una manera simple de asignar la tarea a los trabajadores es dividir los libros equitativamente entre ellos. Cada trabajador apila los libros que le corresponden, uno a la vez. Esta podría no ser una buena división del trabajo, ya que los trabajadores podrían estar todo el tiempo de un lado al otro de la biblioteca. Una forma alternativa de división de trabajo es asignar un conjunto fijo y disjunto de compartimentos a cada trabajador. Si un trabajador encuentra un libro que corresponde a su compartimento lo ubica en su lugar; en caso contrario, lo pasa al responsable del compartimento que corresponda. Este segundo enfoque requiere menos esfuerzo individual.

Este ejemplo sugiere cómo una tarea puede realizarse más rápido dividiéndola en un conjunto de subtareas asignadas a múltiples trabajadores ("*workers*"). Estos cooperan, se pasan los libros cuando es necesario, y completan la tarea. El procesamiento paralelo trabaja precisamente sobre los mismos principios. Dividir una tarea entre trabajadores asignándoles un conjunto de libros es una instancia de *particionamiento de tareas*. El pasaje de los libros es un ejemplo de *comunicación* entre subtareas.

Los problemas son paralelizables en distintos grados. Para algunos, asignar particiones a otros procesadores podría significar mayor consumo de tiempo que realizar el procesamiento localmente. Otros problemas pueden ser completamente secuenciales. Un problema puede tener distintas formulaciones paralelas, lo que puede resultar en beneficios variados, y todos los problemas no son igualmente adecuados para el procesamiento paralelo.

Numerosas y muy variadas áreas presentan problemas que pueden ser resueltos utilizando procesamiento paralelo. Están incluidas las aplicaciones de predicción del clima, monitoreo de contaminación, modelización de biósfera, procesamiento de datos recogidos por satélites, sensado remoto, optimizaciones discretas, modelización oceánica, tomografías computadas, diseño y dinámica de vehículos, análisis de estructuras de proteínas, estudio de fenómenos químicos, procesamiento de imágenes, búsquedas en árboles, sorting de grandes secuencias, exploración petrolera, procesamiento de lenguaje natural, reconocimiento de voz, aprendizaje en redes neuronales, visión por computadora, procesamiento de consultas en bases de datos. Muchos de estos son considerados problemas *Grand Challenge*, esto es, problemas fundamentales en ciencia o ingeniería con un gran impacto económico y científico, y cuya solución puede obtenerse aplicando técnicas y recursos de computación de alta performance.

## 1.2 Definiciones y Conceptos Básicos

Un *proceso* o *tarea* es un bloque de programa secuencial, con flujo de control propio. Es un concepto fundamental de la programación concurrente. Un *procesador* es el dispositivo físico sobre el cual se ejecuta el proceso.

La *concurrencia* define la ejecución simultánea de dos o más procesos en uno o más procesadores.

El *paralelismo* es la ejecución concurrente (esto es, en el mismo instante de tiempo) sobre distintos procesadores. En general, dado un problema, se lo divide en subproblemas que son resueltos simultáneamente por los procesadores.

Los objetivos principales del paralelismo están relacionados con la posibilidad de ajustar el modelo de arquitectura y software al mundo real, y con incrementar la velocidad de resolución de problemas mediante la utilización de varios procesadores, además de mejorar la eficiencia de los mismos.

El paralelismo puede lograrse cuando el sistema de hardware comprende un conjunto de procesadores o elementos de procesamiento vinculados de alguna forma y con capacidad para ejecutar de manera coordinada un algoritmo. Una arquitectura paralela constituye el soporte de hardware que permite obtener concurrencia real, y existe una gran cantidad de variantes que serán tratadas en el Capítulo 2.

Dado que la concurrencia y el paralelismo implican la existencia de varios procesos que interactúan en la resolución de un problema, aparecen conceptos fundamentales que no existen en el "mundo secuencial", como la *comunicación* (para intercambiar datos entre procesos) y la *sincronización* (para evitar interacciones indeseadas).

El *estado* de un proceso está dado por el contenido de sus variables implícitas y

explícitas. A partir de un *estado inicial*, la ejecución de *acciones* provoca transformaciones de estados. La *historia* de un proceso está dada por las acciones que ejecuta en su corrida. Dado que las tareas pueden intercalarse en el tiempo, se deben fijar restricciones. El objetivo de la *sincronización* es restringir las historias de un programa sólo a las deseables. Mediante la posesión de información acerca de otro proceso se coordinan actividades.

Existen dos formas básicas de *sincronización*. La primera es la *exclusión mutua*: consiste en evitar que dos o más procesos se encuentren en su "sección crítica" (parte del código que accede a un recurso compartido) al mismo tiempo. La segunda es *por condición*: a un proceso no se le permite continuar si no se cumple una circunstancia dada. En algunos casos es necesario utilizar ambas técnicas de manera conjunta. Por ejemplo, en el acceso a un *buffer* compartido por un proceso que produce y uno que consume: la exclusión mutua es necesaria para evitar que ambos procesos accedan simultáneamente (uno para dejar y el otro para extraer), y la sincronización por condición para que el productor no deposite cuando el buffer está lleno y el consumidor no intente sacar de un buffer vacío.

La *comunicación* se refiere a cómo organizar y transmitir la información compartida por los procesos concurrentes. Esto es, la manera por la cual un proceso provee datos u obtiene resultados de otro. Los procesos pueden comunicarse principalmente por dos mecanismos: memoria compartida y pasaje de mensajes. Esto se relaciona de alguna forma con la arquitectura de procesamiento utilizada.

El modelo básico de comunicación entre procesos es que éstos accedan a un área de memoria compartida donde consultan o actualizan datos de manera coordinada. Esto obliga a manejar los accesos de modo evitar que los procesos operen simultáneamente sobre los ítems compartidos. Existen diversos métodos para realizar esto, como los semáforos y los monitores [17, 18].

En la comunicación por pasaje de mensajes, existe alguna forma de canal (lógico o físico) a través del cual los procesos envían y reciben información. La comunicación por mensajes pueden ser de tipo sincrónico o asincrónico, uni o bidireccional, punto a punto o broadcast, etc., lo que da lugar a diferentes modelos físicos y semánticos [17, 18].

## 1.3 ¿Qué es un Algoritmo Paralelo?

El diseño de algoritmos es un aspecto fundamental de la Ciencia de la Computación. Cada rama de este campo en algún punto debe tratar con el tema de diseñar un método (algoritmo) que luego se convierte en programa para resolver un problema. Los algoritmos para máquinas convencionales son conocidos como *secuenciales* o *seriales*.

Los *algoritmos paralelos* son métodos para resolver problemas computacionales en máquinas paralelas.

La construcción de algoritmos paralelos significa un desafío ya que, entre otras cuestiones, es necesario tener en cuenta la arquitectura sobre la cual se ejecutará el programa, el modelo de comunicación utilizado, la manera de dividir el problema y los datos, etc. En Secciones posteriores se tratarán estos temas con mayor profundidad.

Al diseñar los algoritmos paralelos, si los procesos no son totalmente independientes y comparten memoria, se deben usar mecanismos de sincronización que agregan complejidad y overhead. Además, las actividades dentro de los programas pueden fallar en estar activas ("vivas"): una o más pueden detenerse por una cantidad de razones (por ejemplo porque otras actividades están consumiendo todos los ciclos de CPU, o porque dos tareas están en *deadlock*). Por otra parte, puede existir no determinismo ya que las actividades son intercaladas ("interleaved") arbitrariamente y dos ejecuciones del mismo programa no necesariamente son idénticas. Esto puede hacer a los programas concurrentes difíciles de predecir, entender y depurar.

No debe perderse de vista el hecho de que un programa paralelo implica la necesidad de asignar (*mapear*) procesos lógicos a procesadores físicos, y que este es un tema de fundamental importancia para el éxito o el fracaso en la resolución eficiente del problema.

## 1.4 Modelos de Concurrencia

### 1.4.1 Memoria Compartida

En este modelo los procesos comparten un espacio de direcciones, por lo que debe asegurarse la sincronización (por exclusión mutua o por condición) para evitar interferencia en el acceso.

El aspecto básico a resolver es el conocido como *problema de la sección crítica (SC)* [17, 18], en el cual se debe resguardar el acceso a los recursos compartidos. Las soluciones deben satisfacer las propiedades de *exclusión mutua* (a lo sumo un proceso está en su SC), *ausencia de deadlock* (si dos o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito), *ausencia de demora innecesaria* (si un proceso trata de entrar a su SC y los otros están en sus secciones no críticas o terminaron, el primero no está impedido de entrar a su SC), y *eventual entrada* (un proceso que intenta entrar a su SC eventualmente lo hará). Entre las herramientas para manejar la concurrencia con memoria compartida se encuentran:

#### Variables Compartidas

Se utilizan variables normales que pueden ser accedidas por los procesos. Esto lleva a complejos protocolos de acceso a la sección crítica, difíciles de diseñar y probar corrección.

Entre los algoritmos más conocidos se encuentran *spin-locks*, *tie-breaker*, *ticket*, *bakery* y sincronización por barreras [17, 18]. Un problema común a todos es que no hay una clara separación entre las variables de sincronización y las usadas para computar resultados, y son ineficientes si los procesos se implementan por multiprogramación.

### Semáforos

Los semáforos fueron una de las primeras herramientas para diseñar protocolos de sincronización. Permiten proteger secciones críticas y pueden usarse para implementar sincronización por condición [17, 18].

Un semáforo es una instancia de un tipo de datos abstracto con sólo 2 operaciones ($P$ y $V$). La operación $V$ señala la ocurrencia de un evento, mientras $P$ se usa para demorar un proceso hasta que ocurra un evento. La utilidad de este tipo de variables está dada en que ambas operaciones son atómicas (esto es, ininterrumpibles). Permiten resolver cualquier problema de sincronización. Entre las desventajas, la naturaleza de bajo nivel de $P$ y $V$ puede llevar a error al utilizarlos, y además la exclusión mutua y la sincronización se proveen mediante las mismas primitivas.

### Regiones Críticas Condicionales

Brindan una notación estructurada para especificar sincronización. Las variables compartidas que necesitan exclusión mutua son declaradas en *recursos*. Las variables en un recurso son accedidas sólo en sentencias *region* que nombran el recurso. Las *region* que nombran el mismo recurso ejecutan con exclusión mutua. La sincronización por condición se logra con condiciones booleanas en las sentencias *region*. Con esto se logra exclusión mutua implícita y sincronización explícita, brindando una mayor facilidad de uso que los semáforos [17, 18].

Las regiones críticas condicionales introdujeron el uso de condiciones de sincronización booleanas, utilizadas luegoen varias notaciones de lenguajes. Imponen restricciones al compilador en el uso de variables compartidas, dando lugar a programas más estructurados y un sistema de prueba más simple pues la interferencia se evita automáticamente. Sin embargo, la implementación de *region* es más cara pues las condiciones de demora deben reevaluarse al cambiar el valor de una variable compartida.

### Monitores

Son módulos de programa que proveen más estructura que las regiones críticas condicionales y pueden implementarse tan eficientemente como los semáforos. Básicamente son un mecanismo de abstracción de datos: encapsulan las representaciones de recursos

abstractos y proveen un conjunto de operaciones que son los únicos medios para manipular la representación [17, 18].

Un monitor contiene variables que almacenan el estado del recurso y procedimientos que implementan operaciones sobre él. La exclusion mutua es implícita ya que la ejecución de procedures en el mismo monitor no se superpone, y la sincronización por condición se brinda a través de un mecanismo de bajo nivel llamado *variables condición*. El programa concurrente queda constituido por procesos activos y monitores pasivos. Dos procesos interactúan llamando procedures en el mismo monitor.

Las variables condición sirven para demorar un proceso que no puede seguir ejecutando en forma segura hasta que el estado satisfaga alguna condición, y para despertar a un proceso demorado cuando la misma se convierte en verdadera. El valor de una variable condición es una cola de procesos demorados, y el mismo no es visible directamente al programador. La condición de demora booleana está asociada implícitamente con la variable por el programador.

La operación *wait* permite demorar un proceso al final de la cola y éste deja el acceso exclusivo al monitor. La operación *signal* despierta al proceso que está al frente de la cola y lo saca de ella (o no tiene efecto si la cola está vacía). Ese proceso ejecuta cuando pueda readquirir el acceso exclusivo al monitor (existen distintas políticas en este sentido).

*Wait* y *signal* son similares a $P$ y $V$, pero hay diferencias: 1) *signal* no tiene efecto si ningún proceso está demorado sobre la variable condición, 2) *wait* siempre demora un proceso hasta un *signal* posterior, y 3) el proceso que hace *signal* siempre ejecuta antes que un proceso despertado como resultado del mismo.

### 1.4.2 Pasaje de Mensajes

El pasaje de mensajes es una extensión de los semáforos para transportar datos y proveer sincronización. Asume la existencia de alguna forma de *arquitectura de red*. Los procesos comparten *canales*. Un *canal* es una abstracción de una red de comunicación física; provee un camino (*path*) de comunicación entre procesos [17, 18].

Los canales son lo único que comparten los procesos, por lo que no se necesita proveer la exclusión mutua; son accedidos por dos clases de primitivas (una para enviar y otra para recibir mensajes), y la sincronización está dada en que un mensaje no puede ser recibido hasta después de ser enviado. Existen variantes en lo que se refiere a los canales y su funcionamiento (globales, conectados a receptores o punto a punto, uni o bidireccionales, sincrónicos o asincrónicos), dando lugar a diferentes notaciones.

### Mensajes Asincrónicos

En este caso, los canales son colas ilimitadas de mensajes enviados y aún no recibidos. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una operación *send*, la cual no bloquea al emisor. Un proceso recibe un mensaje desde un canal mediante un *receive*, que demora al receptor hasta que el canal no esté vacío; luego toma el primer mensaje y lo almacena en variables locales [17, 18].

Proveen una manera sencilla de comunicar procesos. Sin embargo, es necesario un *reply* para confirmar la llegada de un mensaje y la distribución no está garantizada ante fallas. Por otra parte, los mensajes deben ser buffereados y el espacio es finito.

### Mensajes Sincrónicos

Los mensajes sincrónicos resuelven los problemas planteados para los asincrónicos. Tanto la emisión como la recepción son bloqueantes, por lo que existe sincronización en todo punto de comunicación [17, 18].

Los canales son enlaces (*links*) punto a punto entre dos procesos, y las sentencias de entrada y salida son el único medio por el cual los procesos se comunican. El efecto de la comunicación es el de una sentencia de asignación distribuida.

Una sentencia de salida tiene la forma $Destino\ !\ port(e_1, \cdots, e_n)$. Una sentencia de entrada es del tipo $Fuente\ ?\ port(x_1, \cdots, x_n)$. *Destino* y *Fuente* nombran un proceso, mientras *port* es el nombre de un canal entre ambos.

Una sentencia de entrada o salida demora al proceso hasta que otro alcance una sentencia de *matching*; luego las dos sentencias se ejecutan simultáneamente. Dos procesos se comunican cuando ejecutan sentencias de comunicación matching. Una sentencia de salida y una de entrada matchean si todas las partes son compatibles: los procesos se nombran mutuamente y coinciden los nombres de ports y los tipos y cantidades de argumentos y parámetros.

La *comunicación guardada* se utiliza cuando un proceso puede querer comunicarse con más de uno de otros procesos (quizás por distintos ports) y no saber el orden en el cual los otros podrían querer comunicarse con él. En este caso, la guarda está formada por una condición (expresión booleana) y una sentencia de comunicación. La guarda tiene éxito si la condición es verdadera y ejecutar la comunicación no causaría demora (algún otro espera en una sentencia de comunicación matching). La guarda falla si la condición es falsa, y se bloquea si la condición es verdadera pero no puede ejecutarse la comunicación sin causar demora. En muchos casos es útil tener varias sentencias de comunicación guardadas que referencian arreglos de procesos o ports y que difieren solo en el subíndice que emplean.

### 1.4.3 CSP

*Communicating Sequential Processes* (CSP) fue desarrollado por Hoare [168, 169]. En él, un programa puede ser descripto como un conjunto estático de procesos independientes que se comunican. La comunicación es *half-duplex*, y mediante las primitivas *?* y *!* se obtiene un *rendezvous* simple.

Un proceso en CSP se describe en términos de su alfabeto (eventos en que puede incurrir) y su comportamiento. Una comunicación entre procesos es un tipo de evento que puede pertenecer al alfabeto de los mismos, y se expresa matcheando comandos de entrada y salida donde cada proceso nombra al otro (esto es, la comunicación es simétrica).

La sintaxis de los constructores llamador/llamado tiene la forma: *<nombre-proceso-destino> ! <expresión>* y *<nombre-proceso-fuente> ? <variable-destino>*. La primera sentencia debería aparecer en el llamador y la segunda en el proceso llamado. Los valores son copiados desde el llamador al llamado. No se provee *buffering* automático, por lo que la comunicación es sincrónica: los procesos son demorados hasta que se pueda realizar la copia de los valores.

Los comandos de entrada en el proceso llamado pueden aparecer dentro de una "guarda" (con una expresión booleana o sentencia de entrada, pero no una sentencia de salida): la entrada solo se aceptará cuando la condición se satisfaga. Se pueden usar varias guardas con un conjunto de alternativas de entrada, pero sólo una puede seleccionarse y de manera no determinística si más de una es verdadera.

La implementación sugerida de este modelo es para programas ejecutados en un solo procesador, o una red fija de procesadores conectados por canales de entrada/salida, como por ejemplo los transputers. El lenguaje de programación *Occam* fue creado para proveer soporte de programación al transputer [184, 185].

### 1.4.4 DP

*Distributed Processes* (DP) fue creado por Brinch Hansen [51] y es un sucesor de Pascal Concurrente [50]. En DP un programa puede especificarse en términos de un número fijo de procesos concurrentes dispersos en una red de procesadores distribuidos. No hay estructuras de1 datos compartidas, sino que las variables son privadas del proceso que las declara, y sólo pueden ser accedidas por él [17, 18].

La comunicación se provee permitiendo a un proceso llamar a procedimientos comunes definidos dentro de otro proceso. La sincronización se logra a través de "regiones con guarda" (sentencias *when* no determinísticas). Los comandos con guarda se usan para constructores iterativos, de selección y de demoras.

La relación entre procesos no es simétrica: a diferencia de CSP, el proceso llamado no

necesita saber quién lo llamó. La sintaxis de los procedures es tipo Pascal, de la forma *call <nombre proceso>.<nombre procedure>* (*<parámetros>*). La especificación de un procedure dentro de un proceso es similar a Pascal.

Cuando un proceso espera que una condición sea verdadera (por ejemplo, un llamado a uno de sus procedures) el procesador está ocioso. Si más de una condición con guarda es verdadera, la elección de la próxima acción a ejecutarse es no determinística.

Los procesos se usan como módulos de programa en un sistema multiprocesador con memoria local o distribuida, y cada procesador se dedica a una tarea. Los procesos son creados estáticamente, por lo cual la topología es conocida en compilación.

### 1.4.5 RPC

*Remote Procedure Call* (RPC) combina aspectos de monitores y mensajes sincrónicos. Cada componente de programa (módulo) contiene tanto procesos como procedures (operaciones), y los módulos pueden residir en espacios de direcciones distintos. Los procesos de un módulo pueden compartir variables y llamar a procedures declarados él. Un proceso en un módulo puede comunicarse con procesos en otro sólo llamando procedures de éste. Cada operación es un canal de comunicación bidireccional entre el llamador y el servidor. El llamador se demora hasta que la operación llamada haya sido ejecutada y se devuelven los resultados [17, 18].

Cada módulo consta de dos partes: la *especificación* (que incluye *headers* de procedures que pueden ser llamados desde otros módulos) y el *body* (que implementa estos procedures y opcionalmente contiene variables locales, código de inicialización, y procedures locales y procesos). El encabezado de un procedure visible tiene la forma *op opname* (*formales*) *returns result*. El cuerpo de un procedure visible es contenido en una declaración *proc*. Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando *call Mname.opname* (*argumentos*).

Cada vez que se recibe un llamado a un procedure, se crea un nuevo proceso para manejarlo. Se habla de *llamado a procedimiento remoto* pues el llamador y el cuerpo del procedure pueden estar en máquinas distintas. El proceso llamador se demora mientras el proceso *server* ejecuta el cuerpo del procedure que implementa *opname*.

Por sí mismo, RPC es solo un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del servidor, y así la sincronización entre ambos es implícita). Un punto a tener en cuenta es que se necesita que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende exclusión mutua y sincronización por condición, lo cual puede proveerse mediante los mecanismos descriptos para memoria compartida.

### 1.4.6 Rendezvous

RPC provee sólo un mecanismo de comunicación intermódulo, y con frecuencia se necesitan otros procesos para manipular los datos comunicados por medio de RPC. Rendezvous combina las acciones de servir un llamado con otro procesamiento de la información transferida [17, 18].

Un proceso exporta operaciones que pueden ser llamadas por otros. Las declaraciones de operación tienen la misma forma que en los módulos. Como con RPC, un proceso invoca una operación por un *call*, que nombra otro proceso y una operación en el mismo. Pero a diferencia de RPC, la operación es servida por el proceso que la exporta. Por lo tanto, las operaciones son servidas una por vez en lugar de concurrentemente.

Si un proceso exporta una operación *op*, puede entrar en rendezvous con un llamador de *op* ejecutando una sentencia *in* que nombra a la operación. Esta sentencia de entrada es más poderosa que la de mensajes sincrónicos: *in* demora al servidor hasta que haya al menos un llamado pendiente de *op*, luego elige el llamado más viejo, copia los argumentos en los parámetros formales, ejecuta las sentencias, y retorna los resultados. El llamador se demora cuando alcanza un *call*; continúa después de que el server ejecuta la operación llamada. Pero, con rendezvous el server es un proceso activo que ejecuta tanto antes como después de servir una invocación remota.

La sentencia *in* puede generalizarse para combinar comunicación guardada con rendezvous. El lenguaje *Ada* soporta rendezvous por medio de *accept* y comunicación guardada por medio de *select* [266, 40, 157]. Una guarda en una operación guardada tiene éxito cuando: 1) la operación fue llamada, y 2) la expresión de sincronización correspondiente es verdadera. La ejecución de *in* se demora hasta que alguna guarda tenga éxito. Si más de una tiene éxito, elige una no determinísticamente.

En la forma general de rendezvous, puede usarse una *expresión de scheduling* para alterar el orden de servicio de invocaciones por defecto (que es del tipo cola). Si hay más de una invocación que hace que una guarda tenga éxito, entonces la que minimiza el valor de la expresión de scheduling es servida primero.

## 1.5 Modelos de Computación Paralela

Los *modelos* (representación física, matemática o lógica de una entidad real) permiten simular un cierto fenómeno y pueden usarse para describir sistemas conceptuales. En la Ciencia de la Computación, los modelos de computación son usados para describir entidades reales tales como computadoras; como tal, son una versión estilizada de una máquina, capturando características esenciales e ignorando detalles sin importancia de la implementación. Por otra parte, estos modelos son usados como herramientas para

pensar problemas y expresar algoritmos. Aquí, un modelo no se relaciona necesariamente a ninguna computadora real, sino que su principal razón de ser es llevar a entender la *computación*.

El modelo provee un marco para estudiar problemas, obtener ideas sobre sus distintas estructuras, y desarrollar soluciones. Una vez que un algoritmo fue diseñado para resolver un problema con un cierto modelo, éste permite dar una descripción significativa del algoritmo y derivar un análisis preciso.

Los modelos fueron usados desde el inicio de la Ciencia de la Computación, incluso antes de que se nombrara formalmente al campo. Ejemplos de los primeros modelos son los *autómatas*, las *máquinas de Turing*, las *gramáticas formales*, y las *funciones recursivas*. Los más recientes incluyen las *máquinas de acceso aleatorio* (random access machines, *RAM*), las *máquinas paralelas de acceso aleatorio* (parallel random access machines, *PRAM*), *LogP*, *BSP*, etc.

La computación uniprocesador se benefició por la existencia de un modelo teórico simple de computadora uniprocesador (*RAM*). Esto hizo posible desarrollar algoritmos uniprocesador y establecer correctitud de algoritmo y performance esperada en forma relativamente independiente de la máquina específica sobre la cual ejecuta el algoritmo. La optimización para características dependientes de la máquina tales como tamaño de *cache*, número de registros, etc, es manejada por el compilador. Generalmente, a nivel del algoritmo o del programador, uno no tiene en cuenta estas consideraciones. Aunque se propuso un modelo para *RAM* con memorias jerárquicas [1], no se le encontró uso significativo. La simplicidad del modelo *RAM* y su precisión en la modelización de máquinas uniprocesador junto al hecho de que la tecnología de compiladores es suficientemente sofisticada para manejar el gap entre el modelo y las computadoras específicas, hizo posible la computación uniprocesador eficiente [292].

Observando el caso monoprocesador, pueden plantearse algunos requerimientos mínimos que debería cumplir un modelo para computadoras paralelas: ser conceptualmente simple de entender y usar, los algoritmos determinados como correctos por el modelo deben serlo para las arquitecturas de destino, la performance real debe corresponderse con la predicha por el modelo, y ser cercano a las arquitecturas reales para minimizar la brecha entre ambos. Esto es, uno de los objetivos en la definición de un modelo de computación paralela es la posibilidad de predicción de performance que brinde el mismo; el éxito o fracaso dependerá en gran parte de este punto.

Al tratar las máquinas paralelas se encuentran un gran número de modelos abstractos, pero ninguno con la simplicidad y precisión del *RAM*. Además, ninguno intenta servir como modelo para *todas* las clases de máquinas paralelas, y la tecnología de compiladores paralelos no puede manejar el gap entre los modelos y las computadoras comerciales. Las dificultades involucradas en formular un único modelo simple y preciso para computadoras paralelas pueden medirse examinando las variaciones en las máquinas comerciales

y propuestas: hay sincrónicas, asincrónicas y semi-sincrónicas; algunas tienen memoria compartida, otras memoria distribuida, y otras ambas; algunas operan en modo *SIMD* mientras otras en modo *MIMD*; difieren en la red de interconexión usada y/o en los métodos de ruteo; etc.

En las siguientes Secciones se detallan algunos de los modelos más conocidos.

### 1.5.1 *PRAM*

El primer modelo de computación paralela (y el más simple) fue *PRAM* (*P*arallel *R*andom *A*ccess *M*achine) [111, 208, 321]. Simplemente extiende *RAM* permitiendo que varios procesadores compartan la misma memoria.

#### El modelo *RAM*

*R*andom *A*ccess *M*achine (*RAM*) es un modelo secuencial de computación [4, 7, 8]. Consta de:

1. Una memoria con $M$ posiciones. En principio, $M$ puede ser ilimitada (al menos es un número finito arbitrariamente grande). Cada posición de memoria almacena un dato y puede ser accedido aleatoriamente usando una dirección única.
2. Un procesador operando bajo el control de un algoritmo secuencial. Es capaz de cargar y almacenar datos desde y hacia la memoria, y de ejecutar operaciones aritméticas y lógicas básicas. Posee un número constante de registros internos para realizar computaciones sobre datos.
3. Una unidad de acceso a memoria (MAU) que crea un camino desde el procesador a una posición de memoria arbitraria. Cada vez que el procesador necesita leer o escribir en la memoria, provee a la MAU la dirección de la posición; con ésta se establece una conexión directa entre el procesador y la posición de memoria.

Cada paso del algoritmo consta de hasta 3 fases:

1. READ, en la cual el procesador lee un dato desde una posición arbitraria en memoria a uno de sus registros,
2. COMPUTE, donde el procesador realiza una operación básica sobre los contenidos de uno o dos de sus registros, y
3. WRITE, en la que el procesador escribe los contenidos de un registro en una posición arbitraria de memoria.

### El modelo *PRAM*

Consta de [7]:

1. Un número de procesadores idénticos $P_1, \cdots, P_N$ (del tipo usado en $RAM$). En principio, $N$ es ilimitado, o al menos es un número finito arbitrariamente grande.
2. Una memoria común (también del tipo usado en $RAM$) con $M$ posiciones. Nuevamente, $M$ es ilimitada o al menos un número finito arbitrariamente grande tal que $M \geq N$. Esta memoria es compartida por los $N$ procesadores.
3. Una unidad de acceso a memoria (MAU), que permite a los procesadores ganar el acceso a la memoria.

La memoria compartida almacena datos y sirve como medio de comunicación para los procesadores. El modelo permite que cada procesador tenga su propio algoritmo y los operadores trabajan de manera asincrónica. Una aplicación útil de $PRAM$ se da cuando todos los procesadores ejecutan el mismo algoritmo sincrónicamente. Este modo de operación es adecuado para el diseño y análisis de algoritmos eficientes para una cantidad de problemas. Cada paso de un algoritmo para $PRAM$ consta de hasta tres fases:

1. READ, en la cual (hasta) $N$ procesadores leen simultáneamente desde (hasta) $N$ posiciones de memoria. Cada procesador lee a lo sumo una posición y almacena el valor en un registro local,
2. COMPUTE, donde (hasta) $N$ procesadores realizan operaciones aritméticas o lógicas básicas sobre sus datos locales, y
3. WRITE, en la que (hasta) $N$ procesadores escriben simultáneamente en (hasta) $N$ posiciones de memoria. Cada procesador escribe el valor contenido en un registro local en a lo sumo una posición.

La frase "(hasta) $N$ procesadores" significa que por medio de control algorítmico, algunos pueden ser prevenidos de ejecutar un paso dado, ya que cada procesador tiene un índice único y ese valor puede usarse para habilitarlo o deshabilitarlo.

Una computación $PRAM$ comienza con un *input* almacenado en memoria global y un único procesador activo. En cada paso de la computación, un procesador activo y habilitado puede leer un valor desde una posición de memoria local o global, ejecutar una única operación $RAM$ y escribir en una posición de memoria local o global. Alternativamente, un procesador puede activar a otro. La computación termina cuando todos concluyen su operación [275].

El costo de una computación *PRAM* es el producto de la complejidad del tiempo paralelo y el número de procesadores usados. Por ejemplo, un algoritmo *PRAM* de complejidad de tiempo $O(log\, p)$, usando $p$ procesadores tiene costo $O(p * log\, p)$ [278].

Dado que no prevé las variaciones, no puede modelizar todas las computadoras paralelas sincrónicas con precisión. De hecho, aún para modelizar máquinas paralelas sincrónicas de memoria compartida, se necesitan considerar variantes que toman en cuenta las diferencias en los esquemas de resolución de conflictos de acceso a memoria. Hay distintas maneras para que los procesadores accedan a memoria, las cuales son posibles por medio del repertorio de instrucciones de *PRAM*. Estas instrucciones para leer y escribir son *Exclusive Read (ER)*, *Concurrent Read (CR)*, *Exclusive Write (EW)* y *Concurrent Write(CW)*. Esta última puede refinarse para especificar qué queda almacenado en una posición dada cuando varios procesadores intentan escribir en ella simultáneamente. Con esto se logran extensiones para ser usadas con *CW*, como *Priority CW*, *Common CW* (y sus variantes *Fail Common*, *Collision Common* y *Fail Safe Common*), *Arbitrary CW*, *Random CW* y *Combining CW*. Esto da lugar a los modelos EREW-*PRAM*, CREW-*PRAM* y CRCW-*PRAM*.

La relación entre distintos modelos *PRAM* básicos se discute en [101]. Otras variantes fueron introducidas para modelizar computadoras paralelas de memoria compartida asincrónicas (*APRAM* [71]) y semiasincrónicas (*PPRAM* o phase *PRAM* [124]).

### 1.5.2 *BSP*

Dado que la mayoría de las máquinas paralelas comerciales son un conjunto de pares procesador-memoria que operan asincrónicamente y se comunican vía una red de interconexión, hubo esfuerzos por desarrollar modelos para estas computadoras más precisos que el *PRAM* asincrónico.

El modelo *Bulk Synchronous Parallel* (*BSP*), propuesto por Valiant en 1990 [350], es un intento en este sentido (otro es *LogP*, descripto en la Sección 1.5.3). El propósito es permitir el desarrollo de software escalable e independiente de la arquitectura y brindar un marco de trabajo simple para computaciones paralelas de propósito general. Entre sus características se encuentran el tratamiento del medio de comunicación como una red abstracta completamente conectada y un modelo de costo de sincronización y comunicación independiente y explícito [275].

*BSP* establece un nuevo estilo de programación paralela para programas de propósito general, donde los programas son sencillos de escribir (casi tan simples como los secuenciales), independientes de la arquitectura subyacente (brindando mayor portabilidad), y de performance predecible. Esto se logra elevando el nivel de abstracción en la escritura de los programas. Para ello deben determinarse las propiedades *bulk* de un programa y la habilidad *bulk* de una máquina particular para satisfacerlas. Una manera en que *BSP*

logra abstracción es renunciando a la localidad de programas para optimizar su performance. Esto no puede hacerse en los dominios en que la localidad es crítica, como por ejemplo algunas áreas del procesamiento de imágenes.

La idea fundamental es que *BSP* divide la computación y la comunicación. Es en realidad un modelo semi-asincrónico en el cual los procesadores (o subconjuntos de ellos) trabajan asincrónicamente $L$ unidades de tiempo y luego son sincronizados por algún mecanismo de hardware. Un programa paralelo es una secuencia de *superpasos*, donde cada uno se compone de tareas computacionales y de ruteo. Durante un superpaso, los procesadores pueden realizar computación y enviar y recibir mensajes de otros. Las actividades dentro de un superpaso se realizan asincrónicamente. La sincronización se realiza luego de cada superpaso. La computadora paralela comienza el primer superpaso y luego chequea su completitud después de $L$ unidades de tiempo. En caso de que esté completo, se inicia el próximo. Sino, se asignan otras $L$ unidades al primer superpaso y se espera $L$ unidades de tiempo para testear la completitud. Este proceso se repite hasta que se complete. Los superpasos sucesivos son manejados de la misma manera.

La comunicación interprocesador es manejada por un router no sensible a la topología de red que realiza $h$-relaciones (en una $h$-relación, cada procesador envía a lo sumo $h$ mensajes y recibe a lo sumo $h$ mensajes). El costo de realizar una $h$-relación es $gh$ donde $g$ es una medida del ancho de banda de la red. A causa de la naturaleza semi-asincrónica del modelo, el costo real de una $h$-relación es $[gh/L]$ superpasos.

Una computadora *BSP* queda caracterizada por los siguientes parámetros: el ancho de banda de la red de interconexión, el número de procesadores, sus velocidades y el tiempo de sincronización de los procesadores. Entre los sistemas actuales que conforman el modelo *BSP* se pueden incluir las máquinas monoprocesador, las redes de estaciones de trabajo conectadas con librerías de pasaje de mensajes como MPI [322] o PVM [117], los procesadores de memoria distribuida (IBM SP2, Meiko, Intel Paragon), los procesadores de memoria compartida distribuida (Sillicon Origin, Cray T3E, Convex Spp), y multiprocesadores Pentium [275].

*BSP* está más cercano que cualquiera de las variantes de *PRAM* para modelizar computadoras asincrónicas de memoria distribuida. Logra mayor precisión (o realismo) manteniendo la simplicidad a expensas de ser insensible a la topología de red. Desafortunadamente, esto lo hace incapaz de tomar en cuenta diferencias de programación atribuibles a variantes en la topología. Estas diferencias se vuelven importantes cuando se modelizan computaciones que requieren transferencia simultánea de mensajes entre varios pares de procesadores. No soporta directamente memoria compartida; esto se puede obtener por simulaciones de algoritmos *PRAM* sobre una computadora *BSP*, pero no permite explotar la localidad de datos pues *PRAM* no tiene memoria local. El modelo *BSPRAM* concilia la explotación de la localidad de datos con la programación al estilo de memoria compartida [292, 275].

### 1.5.3 *LogP*

El objetivo de *LogP* es proponer un modelo de computación paralela que sirva como base para el análisis y diseño de cualquier algoritmo con el fin de implementarlo sobre un amplio conjunto de arquitecturas paralelas. El modelo, propuesto por Culler *et. al.* [82] intenta ser realista, teniendo en cuenta los principales factores de performance de los procesadores y la red de interconexión. Un punto de partida para *LogP* fue *BSP*, en el sentido de su visión sobre un modelo realista y sencillo de usar, y de permitir el diseño de algoritmos que funcionen bien sobre un amplio conjunto de máquinas. *LogP* caracteriza una máquina paralela por un conjunto reducido de parámetros, y oculta características específicas como la topología y los algoritmos de ruteo [275]. Desafortunadamente, ignorar lo específico de la red puede resultar en el desarrollo de algoritmos que funcionan bien sobre el modelo pero no sobre la computadora de destino [292].

*LogP* fue desarrollado para una máquina de memoria distribuida con procesadores comunicándose por mensajes punto a punto; tiene en cuenta las características de capacidad de la red pero no su estructura interna de conexión. En este modelo, una red de procesadores asincrónicos es modelizado por los siguientes parámetros:

1. $L$ es una cota superior de la *latencia*, el tiempo máximo necesario para que un mensaje pequeño $M$ viaje entre dos módulos procesador/memoria.

2. $o$ es el *overhead*, definido como el tiempo en que un procesador está transmitiendo o recibiendo un mensaje. Durante el mismo, no puede realizar otras actividades.

3. $g$, el *gap*, es el intervalo mínimo de tiempo entre la transmisión o recepción de dos mensajes consecutivos de tamaño $M$ por el mismo procesador. El inverso de $g$ es el ancho de banda por procesador.

4. $P$, el número de módulos procesador/memoria. Se asume una unidad de tiempo para las operaciones locales (ciclo).

$L$, $o$, y $g$ se miden como múltiplos del ciclo de procesador. El modelo es *asincrónico*, es decir que los procesadores trabajan asincrónicamente y la latencia experimentada por cualquier mensaje es impredecible, pero acotado superiormente por $L$ en ausencia de demoras. A causa de las variaciones en la latencia, los mensajes dirigidos a un módulo destino pueden no arribar en el mismo orden en que se enviaron. El modelo básico asume que todos los mensajes son de tamaño *chico*.

Una de las características distintivas de *LogP* es considerar la superposición de comunicación y computación: mientras los mensajes viajan en la red, el procesador puede realizar computaciones útiles sin esperar a que el mensaje llegue a destino.

La red de interconexión se asume con *capacidad finita*: a lo sumo $\lceil L/g \rceil$ mensajes pueden estar en tránsito desde un procesador a otro en cualquier momento. Si un procesador intenta transmitir un mensaje que excedería este límite, se demora hasta que pueda ser enviado sin exceder el límite de capacidad. Para obtener el valor de estos parámetros, el modelo no tiene en cuenta factores como la saturación de la red, mensajes largos, hardware especial para el ruteo y los patrones de comunicación, todos los cuales pueden afectarlos. El modelo *LogGP*, (Alexandrov *et al.* [10]), incorpora la posibilidad de manejar mensajes más largos incluyendo el parámetro $G$ (tiempo para enviar cada *byte*).

En [234], Li *et al.* compilaron una lista de modelos de computadora paralela y los caracterizaron mediante métricas de recurso (grado de asincronicidad, organización de memoria, latencia, ancho de banda, etc.). Su lista incluye 14 modelos *PRAM*, *VPRAM* (vector *VRAM*), *LPRAM* (*PRAM* de memoria Local), *BPRAM* (block *PRAM*), *PPRAM*, *PLPRAM* (phase LPRAM), *APRAM*, *BSP*, *LogP*, *PMH* (parallel memory hierarchy), *P-HMM* (parallel hierarchical memory model), *H-PRAM* (hierarchical PRAM), y *LogP-HMM*. Otros modelos pueden encontrarse en [262, 318].

### 1.5.4 *LDA*

*Latency-of-Data-Access* (*LDA*) [316] es un modelo de computación realista de arquitecturas paralelas existentes y futuras. Captura el hecho de que la performance de computación está principalmente limitada por la velocidad del sistema de memoria. Los movimientos de datos son considerados como accesos directos a ciertos niveles de la jerarquía de memoria. La cantidad de movimientos, combinada con los costos de los cálculos llevan al tiempo de ejecución de una aplicación. Además, las latencias pueden usarse para determinar la contención causada por tareas concurrentes.

Este modelo pone atención al hecho de que el tiempo de ejecución de las aplicaciones científicas está más limitado por la velocidad de los sistemas de memoria que por la performance de las unidades de procesamiento. *BSP* y *LogP* asumen costos uniformes para todos los accesos a memoria excepto para las comunicaciones. En cambio, el modelo *Memory Hierarchy* (*MH*) considera a la memoria de una máquina secuencial como una secuencia de módulos de memoria acoplados por buses que trabajan concurrentemente. El modelo *Uniform Memory Hierarchy* (*UMH*) [11] reduce la complejidad de *MH*, pero aún mantiene interacciones complejas entre los niveles de la jerarquía de memoria considerados.

### 1.5.5 *BDM*

*Block Distributed Memory* (*BDM*) [189, 25] es usado sobre máquinas de memoria distribuida. Permite el diseño de algoritmos usando un espacio de direcciones único y no asume ninguna topología de interconexión particular. Captura la performance incorpo-

rando una medida de costo para comunicación interprocesador inducida por accessos a memoria remota. Esta medida incluye parámetros que reflejan la latencia de memoria, ancho de banda de comunicación y localidad espacial. El modelo permite ubicación inical de datos y prefetching.

### 1.5.6 *CCM*

El *Modelo de Computación Colectiva* (*CCM*) [132, 294] es una variante de *BSP*, y describe un sistema explotado a través de una plataforma de software standard con facilidades para la creación de grupos, el uso de operaciones colectivas y de memoria remota [275]. Es una generalización formal de modelos que permitan predecir de manera confiable el comportamiento de un conjunto de funciones de comunicación.

En *CCM* la computación ocurre en superpasos, que pueden ser de tipo *normal* (en el cual las tareas de un grupo realizan una actividad secuencial y, de ser necesario, una función colectiva de comunicación) o de *división* (donde la máquina corriente o grupo de procesadores se puede dividir en un conjunto de submáquinas a causa de una función de partición). Para usar *CCM* como modelo de predicción de performance debe asignarse una función de costo, en base al parámetro tiempo, a las funciones de comunicación y división. El modelo es independiente de la plataforma de implementación y predice los tiempos con precisión.

### 1.5.7 Comentarios

Cada modelo intenta proveer una abstracción para desarrollar algoritmos y programas en una clase de computadoras paralelas. La abstracción es generalmente más simple para trabajar que cualquier instancia de la clase modelizada. Pero, esta simplificación se obtiene a expensas de introducir imprecisiones en la modelización. Dada la incapacidad de los compiladores paralelos para compensar esta imprecisión, los algoritmos desarrollados para el modelo pueden resultar en código ineficiente sobre la computadora de destino. Como resultado, la aplicabilidad del modelo está limitada.

La cantidad de modelos muestra que no existe consenso. Evaluados separadamente ninguno es totalmente aceptable, pero en conjunto todos parecen poner énfasis y coincidir en un número de características, por lo que podría encontrarse un modelo consistente. Las características incluyen parámetros como paralelismo computacional, latencia y overhead, ancho de banda, sincronización, jerarquías de memoria y topologías. Estas características reflejan la perspectiva de un objetivo de diseño eficiente de algoritmos. Un modelo unificado debe incluir características que representen los objetivos de programadores, diseñadores y constructores. Tanto los diseñadores de software como de hardware tienen motivaciones para concentrarse en parámetros para medir la performance de los diseños [275].

# Capítulo 2

# Modelos de Arquitecturas Paralelas

## 2.1 Introducción

Las máquinas secuenciales tradicionales se basan en el modelo introducido por John von Neumann, el cual consta de una unidad central de procesamiento (CPU) y memoria. Toma una secuencia única de instrucciones y opera sobre un único flujo de datos [131]. La velocidad de estas máquinas se encuentra limitada por dos factores: la velocidad de ejecución de instrucciones y de los intercambios de información entre memoria y CPU. Esta última puede incrementarse aumentando el número de canales sobre los cuales los datos pueden ser accedidos simultáneamente; esto se realiza dividiendo la memoria en bancos accesibles de manera independiente (*memory interleaving*).

Otra posibilidad es usar memorias intermedias, de menor capacidad y más rápidas, que actúen como "buffers" (*memoria cache*), usando el principio de que si una palabra es accedida desde un lugar de memoria, es probable que los siguientes accesos sean a palabras vecinas.

La velocidad de ejecución de instrucciones puede incrementarse también superponiendo la ejecución de una instrucción con la operación de búsqueda de la próxima. De esta forma, mientras la CPU está ocupada ejecutando la instrucción corriente, la próxima se trae desde la memoria a la cola de instrucciones (*pipelining de instrucción*). Una técnica relacionada es el *pipelining de ejecución*, donde se permite que múltiples instrucciones estén en varias etapas de ejecución en unidades funcionales como multiplicadores y sumadores. La Figura 2.1 muestra las variantes expresadas.

Se han propuesto diversas formas de organizar las arquitecturas de procesamiento paralelo. El problema al definir qué es una arquitectura paralela, y además tener una taxonomía, radica en la gran cantidad de características que deben considerarse, y a que no todas son de fácil descripción, comparación y clasificación.

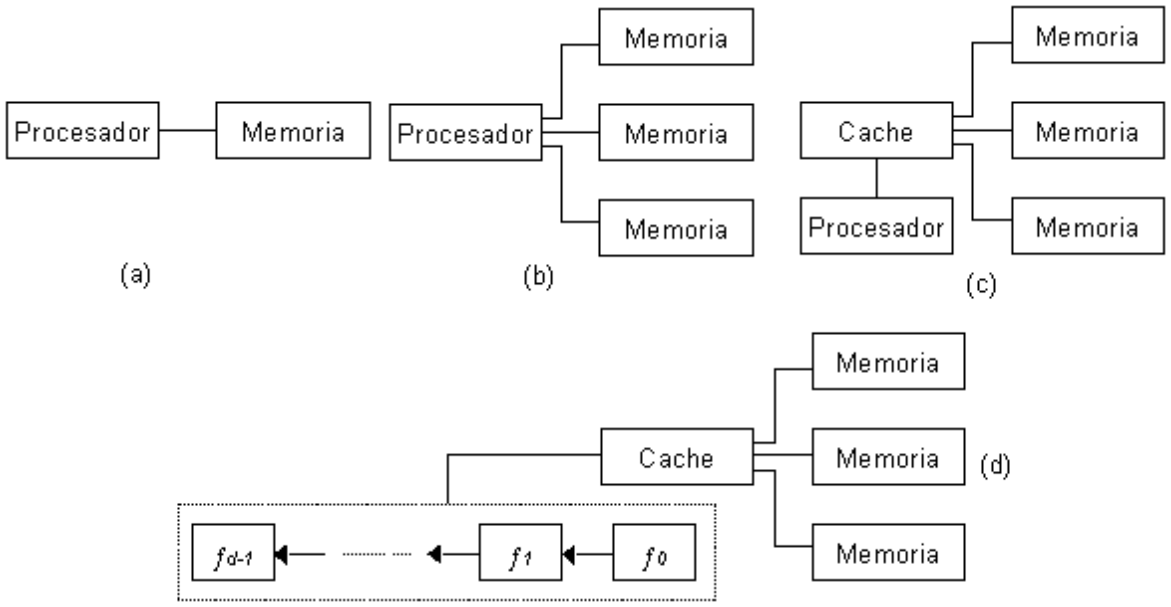


Figura 2.1: Evolución de una computadora secuencial. (a) Simple; (b) Con interleaving de memoria; (c) Con interleaving de memoria y cache; (d) Procesador pipelined con $d$ etapas.

Un sistema *multiprocesador* es una máquina de procesamiento paralelo consistente en un conjunto de procesadores homogéneos para la ejecución de aplicaciones [261]. Puede incluir otros procesadores heterogéneos para procesamiento especial tal como entrada/salida e interfase con periféricos. Los procesadores pueden tener memoria local, memoria compartida o una combinación de ambos. La computadora generalmente es un mini o mainframe, y el número de elementos de procesamiento varía de pocos a miles (en este último caso se habla de sistemas *masivamente paralelos*).

Un *multicomputador* es un sistema distribuido que consta de un conjunto de procesadores paralelos con comunicación interprocesador. Cada procesador tiene memoria local y puede compartir memoria global con algunos de los otros. Típicamente los procesadores son de arquitecturas heterogéneas.

Las computadoras paralelas difieren en varias dimensiones tales como mecanismo de control, organización del espacio de direcciones, granularidad de procesadores y red de interconexión, y por estos criterios es que se clasifican en las siguientes Secciones.

## 2.2 Clasificación de acuerdo al mecanismo de control

La clasificación propuesta por Flynn [110] está centrada en la manera en que las instrucciones son ejecutadas sobre los datos [182, 344]. Cualquier computadora opera ejecutando

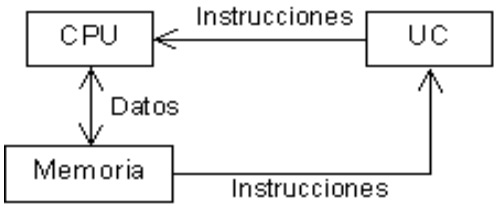


Figura 2.2: Arquitectura SISD

instrucciones sobre datos. Un flujo (*stream*) de instrucciones indica qué hacer en cada paso del procesamiento. Un flujo de datos de entrada son los procesados por estas instrucciones. Pueden identificarse 4 clases básicas de acuerdo a la cantidad de estos elementos presentes en el sistema: *SISD* (Single Instruction stream, Single Data stream), *MISD* (Multiple Instruction stream, Single Data stream), *SIMD* (Single Instruction stream, Multiple Data stream) y *MIMD* (Multiple Instruction stream, Multiple Data stream).

### 2.2.1 SISD

Es la arquitectura usada por la mayoría de los uniprocesadores, y representa básicamente la máquina de von Neumann. Las instrucciones son ejecutadas una después de otra, una por ciclo de instrucción, y la memoria afectada sólo es usada para esa instrucción. Este concepto puede expandirse de acuerdo a lo expresado en la Sección 2.1.

La Figura 2.2 muestra un esquema de esta arquitectura. La Unidad Central de Proceso (CPU) ejecuta las instrucciones (decodificadas por la unidad de control (UC)) sobre los datos. La memoria recibe y almacena los datos en las escrituras, y brinda los datos en las lecturas.

### 2.2.2 MISD

En este caso, un número de procesadores ejecutan un flujo de instrucciones distinto, pero comparten datos comunes. El mismo conjunto de datos es operado simultáneamente por distintos streams de instrucción. Los procesadores operan sincónicamente (en *lock-step*) de modo que los items de datos son manejados en elementos de procesamiento adyacentes durante el mismo ciclo de instrucción. Este esquema puede verse en la Figura 2.3.

Es una arquitectura paralela muy específica, que se adapta sólo a una clase de problemas y no es de propósito general.

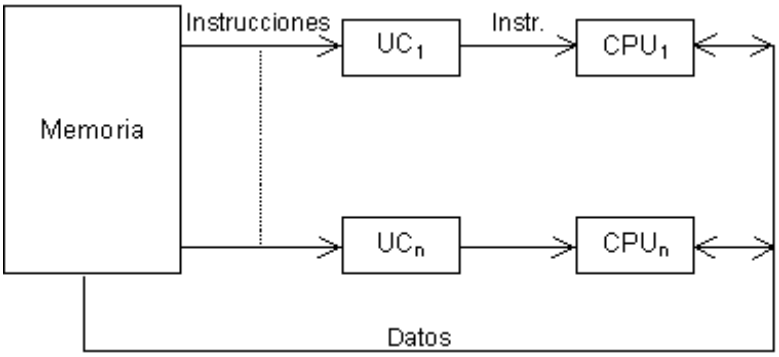


Figura 2.3: Arquitectura MISD

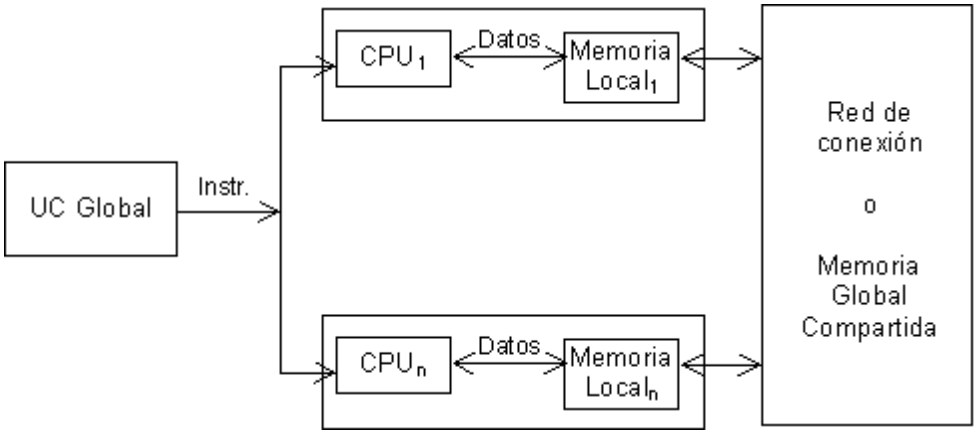


Figura 2.4: Arquitectura SIMD

### 2.2.3 SIMD

Cuenta con un conjunto de procesadores idénticos con sus propias memorias, que ejecutan la misma instrucción bajo el control de una única unidad de control o procesador host, sobre distintos datos. Puede verse en la Figura 2.4. El host hace broadcast de la instrucción a ser ejecutada a los procesadores paralelos simultáneamente; esta ejecución es sincrónica.

En algunos casos se pueden habilitar o deshabilitar selectivamente algunos elementos de procesamiento, con el fin de que ejecuten o no la próxima instrucción. Los módulos procesador-memoria pueden estar comunicados a través de una memoria global compartida o una red de interconexión. Los *array processors*, consistentes de $n$ elementos de procesamiento homogéneos, son multiprocesadores típicos que usan este tipo de arquitecturas. Ejemplos de máquinas de tipo SIMD son Illiac IV, MPP, DAP, CM-2, MasPar MP-1 y MasPar MP-2.

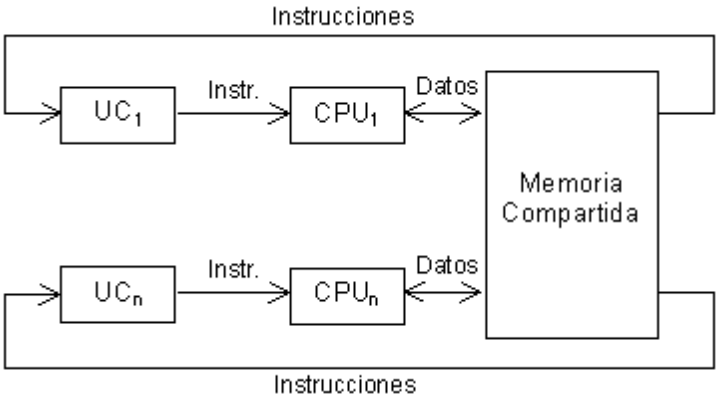


Figura 2.5: Arquitectura MIMD con memoria compartida

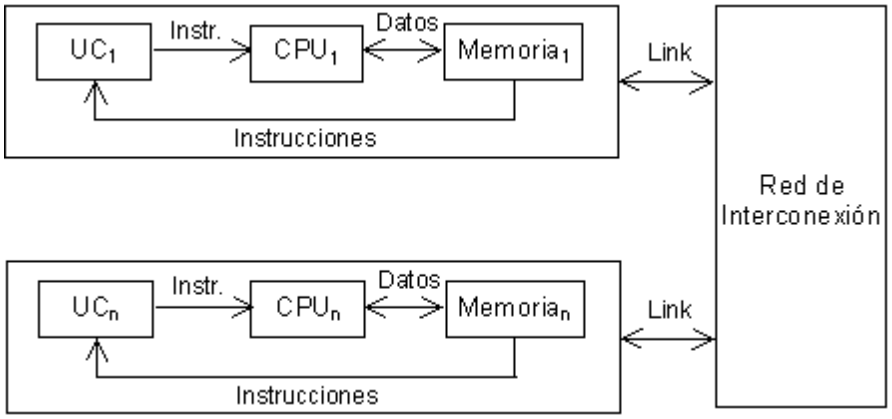


Figura 2.6: Arquitectura MIMD con memoria distribuida

### 2.2.4 MIMD

Cada uno de un número de procesadores tiene su propio flujo de instrucciones y de datos, por lo que puede ejecutar su programa específico independientemente de los otros.

Son las máquinas paralelas de propósito más general. Constan de $n$ procesadores con su propia unidad de control; cada procesador ejecuta una instrucción distinta sobre diferentes datos. Puede existir una memoria compartida (multiprocesadores MIMD con memoria compartida, o *fuertemente acoplados*) como muestra la Figura 2.5. O puede haber memoria local en los procesadores (obteniéndose una memoria distribuida), y una red de interconexión, como se ve en la Figura 2.6. En este caso se habla de multiprocesadores MIMD con memoria distribuida, o *débilmente acoplados*, o multicomputadoras, o computadoras paralelas de memoria distribuida (DMPC).

Ejemplos de computadoras MIMD incluyen Cosmic Cube, nCUBE 2, iPSC, Symme-

try, FX-8, FX-2800, TC-2000, CM-5, KSR-1 y Paragon XP/S [209]. Otro ejemplo son las máquinas *data flow*, como la Manchester Prototype Dataflow Computer [142], y las desarrolladas en UC Irvine [22] y MIT [89]. Ofrecen un alto grado de paralelismo pero son más difíciles de programar que las máquinas *control flow*.

### 2.2.5 Comentarios

Si bien las clasificaciones reconocen un mismo origen [110], en la literatura no siempre se describen las clases de la misma manera. Esto, en algunos casos, puede llevar a que una máquina aparezca en diferentes clases de acuerdo al autor.

Las máquinas SIMD requieren menos hardware que las MIMD (sólo tienen una unidad de control global) y también menos memoria (sólo necesita almacenarse una copia del programa). Son adecuadas para programas con paralelismo de datos (*data parallel*), donde el mismo conjunto de instrucciones se ejecuta sobre un gran conjunto de datos. Como desventaja, en las SIMD distintos procesadores no pueden ejecutar instrucciones diferentes en el mismo ciclo de reloj; por ejemplo en una sentencia condicional, debe ejecutarse secuencialmente el código para cada condición.

Los procesadores individuales en una máquina MIMD son más complejos, porque cada uno tiene su unidad de control. Podría parecer que el costo es mayor que el de un SIMD, pero es posible usar microprocesadores de propósito general como unidades de procesamiento en máquinas MIMD. Por otra parte, la CPU usada en SIMD debe ser diseñada especialmente. Luego, los procesadores en computadoras MIMD pueden ser más baratos y poderosos que en SIMD.

Las SIMD ofrecen sincronización automática entre procesadores después de cada ciclo de instrucción, por lo que son adecuadas para programas paralelos que requieren sincronización frecuente. Algunas MIMD tienen hardware extra para permitir operar también en modo SIMD (por ejemplo, DADO y CM-5).

## 2.3 Clasificación por organización del espacio de direcciones

La resolución de un problema sobre un ensamble de procesadores requiere interacción. Las arquitecturas de *pasaje de mensajes* y las de *espacio de direcciones compartido* (o de *memoria compartida* brindan dos medios diferentes para la interacción interprocesador.

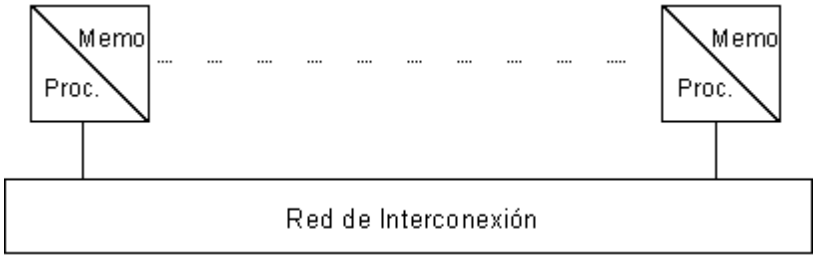


Figura 2.7: Arquitectura de pasaje de mensajes

### 2.3.1 Arquitecturas de Pasaje de Mensajes

En este caso, los procesadores están conectados usando una red de interconexión de pasaje de mensajes. Cada procesador tiene su propia memoria llamada *memoria local* o *memoria privada*, la cual es accesible sólo por él. Los elementos de procesamiento pueden interactuar únicamente por pasaje de mensajes. La Figura 2.7 muestra una arquitectura típica de esta clase. Esta modelo también suele referirse como de *memoria distribuida*. Pueden encontrarse ejemplos en máquinas paralelas como Cosmic Cube, Paragon XP/S, iPSC, CM-5 y nCUBE 2.

### 2.3.2 Arquitecturas de Espacio de Direcciones Compartido

Proveen soporte de hardware para lectura y escritura de todos los procesadores a un espacio de direcciones compartido. Los elementos de procesamiento interactúan modificando objetos de datos almacenados en esa memoria.

Pueden diferenciarse varios modelos de acuerdo a cómo es accedida esta memoria compartida. En el modelo UMA (*Uniform Memory-Access*), donde la memoria física es compartida de manera uniforme por todos los procesadores, todos tienen igual tiempo de acceso a todas las posiciones (Figura 2.8). Una desventaja de este modelo es que el ancho de banda de la red de interconexión debe ser sustancial de modo de asegurar una buena performance. Si todos acceden de la misma forma a los dispositivos periféricos se habla de *Multiprocesador Simétrico*; por el contrario, en un *Multiprocesador Asimétrico* sólo un conjunto de procesadores pueden ejecutar el sistema operativo completo y manejar la entrada/salida [23, 344].

En el modelo NUMA (*Nonuniform Memory-Access*) el tiempo de acceso a memoria depende de la posición a la cual se acceda. En algunos casos, lo que se tiene es una combinación de memorias locales y globales; en otros sólo memorias locales que son accesibles a todos vía una red de interconexión (Figura 2.9).

COMA (*Cache-Only-Memory-Access*) puede verse como un caso especial de NUMA,

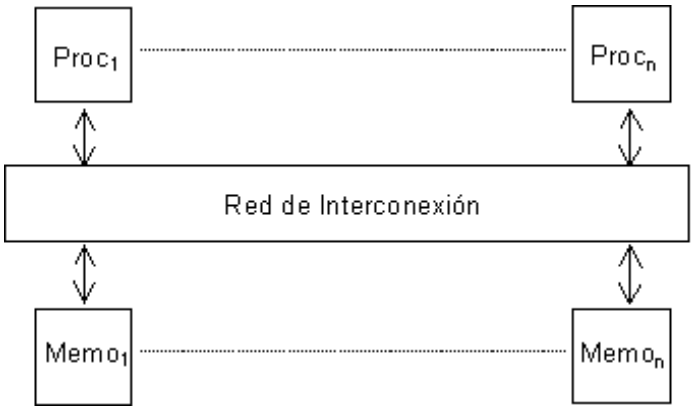


Figura 2.8: Arquitectura de espacio de direcciones compartido UMA

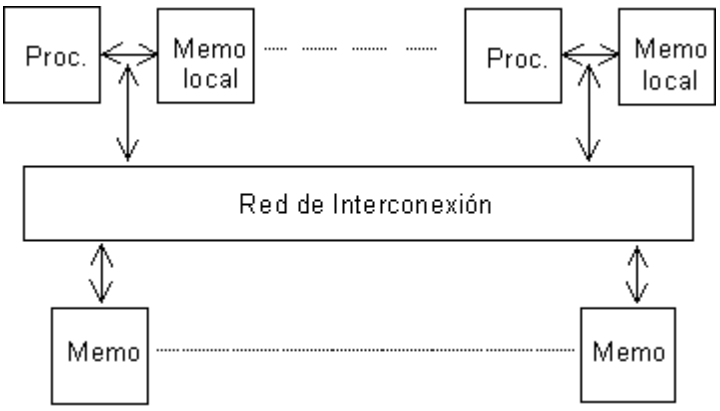


Figura 2.9: Arquitecturas de espacio de direcciones compartido NUMA

donde las memorias compartidas principales son *cache* y forman un espacio global. Los accesos a cache remotos son resueltos por directorios de cache distribuidos, agregando complejidad (y costo) de hardware.

### 2.3.3 Comentarios

Puede notarse que la arquitectura NUMA es similar a la de pasaje de mensajes (la memoria está físicamente distribuida en ambos). La mayor diferencia es que NUMA posee soporte de hardware para acceso de lectura y escritura a las memorias de otros procesadores, mientras en las arquitecturas de pasaje de mensajes el acceso remoto debe simularse explícitamente. Por razones históricas, NUMA es referida como arquitectura de memoria compartida debido a su espacio de direcciones compartido. Ejemplos de NUMA son TC-2000, Stanford Dash y SGI Origin. La KSR-1 es un ejemplo de COMA.

Es sencillo emular una arquitectura de pasaje de mensajes con $n$ procesadores en una computadora de memoria compartida con $n$ procesadores. Puede hacerse particionando la memoria en $p$ bloques disjuntos y asignando una partición a cada procesador; el envio de mensajes se simula haciendo que un procesador escriba en la partición de otro.

El caso inverso es más costoso, pues acceder la memoria de otro procesador requiere enviar y recibir mensajes. Luego, las computadoras de memoria compartida brindan mayor flexibilidad en la programación. Además, algunos problemas requieren rápido acceso por todos los procesadores a grandes estructuras de datos que pueden cambiar dinámicamente, y este acceso es soportado de mejor forma por arquitecturas de memoria compartida. Sin embargo, el hardware para proveer tal espacio compartido tiende a ser más caro que el de pasaje de mensajes.

## 2.4 Clasificación por la granularidad de los procesadores

Una máquina paralela puede estar compuesta de un número chico de procesadores muy poderosos o de un gran número de procesadores relativamente menos potentes. Las computadoras de la primera clase son llamadas *de grano grueso* (*coarse-grain*), mientras que las del segundo tipo se denominan *de grano fino* (*fine-grain*).

Las computadoras de grano grueso tales como Cray Y-MP brindan un número pequeño de procesadores (8 o 16) cada uno con capacidad de muchos GigaFlops (1 Gigaflop = 1 millón de operaciones de punto flotante por segundo). En las de grano fino, tales como CM-2, MasPar MP-1 y MasPar MP-2, existen un gran número de procesadores relativamente lentos (hasta 65536 procesadores de 1 bit en CM-2, hasta 16384 de 4 bits

en MasPar MP-1). Entre los extremos están las computadoras de *grano medio* (*medium-grain*), como la CM-5, nCUBE2 y Paragon XP/S, con unos pocos miles de procesadores, cada uno brindando performance del tipo workstation.

Las diferentes aplicaciones son adecuadas para una u otra clase en distintos grados. Muchas sólo tienen concurrencia limitada, por lo que no pueden hacer uso efectivo de demasiados procesadores y son más adecuados para computadoras de grano grueso. Las máquinas de grano fino son más efectivas en costo para aplicaciones con alto grado de concurrencia. Por lo tanto, debe considerarse un tradeoff entre costo y utilidad de la máquina al elegir la granularidad del procesador.

La granularidad de una computadora paralela puede definirse como el cociente del tiempo necesario para una operación básica de comunicación y el tiempo requerido para una computación básica. Las máquinas para las que este cociente es chico son adecuadas para algoritmos con comunicación frecuente, es decir, aquellos donde el tamaño de grano de la computación (antes de necesitar una comunicación) es chico. Dado que estos algoritmos contienen paralelismo de grano fino, con frecuencia estas máquinas son llamadas computadoras de grano fino. Por el contrario, aquellas en que el cociente es grande son adecuadas para algoritmos que no requieren comunicación frecuente (computadoras de grano grueso).

## 2.5 Clasificación de acuerdo a la red de interconexión

Las computadoras de espacio de direcciones compartido y de pasaje de mensajes pueden construirse conectando unidades de procesador y memoria usando una diversidad de redes de interconexión, que pueden ser clasificadas como *estáticas* o *dinámicas*.

Las redes estáticas constan de enlaces (*links*) de comunicación punto a punto entre procesadores y también suelen nombrarse como redes *directas*. Se usan típicamente para construir computadoras de pasaje de mensajes. Las redes dinámicas están construidas usando *switches* y enlaces de comunicación. Los links están conectados unos a otros dinámicamente por los elementos de switching para establecer caminos entre procesadores y bancos de memoria. También son llamadas redes *indirectas* y normalmente se usan para máquinas de espacio de direcciones compartido. Puede encontrarse una descripción detallada de ambas clases en [209, 7, 344].

Existen una serie de preguntas que uno debe responderse al diseñar la red de interconexión [7]:

1. Qué forma debería tener la red? Esto es, cuántos vecinos debe tener cada procesador, cómo se eligen estos vecinos, todos los procesadores tienen la misma cantidad? La respuesta está dada por la *topología* de la red.

2. Un procesador puede comunicarse con todos sus vecinos a la vez?
3. Cuál es el tamaño de un mensaje que puede transmitir un procesador en determinado tiempo? Es decir, cuántos datos pueden enviarse en una transmisión?
4. Cuánto tarda un procesador en iniciar una transmisión? Este tiempo es significativo?
5. Cuánto tarda un dato en viajar entre dos procesadores vecinos? Este tiempo es función de la longitud del enlace?
6. Cuánto tarda un procesador en recibir un dato? Es significativo?
7. Cuánto demora leer o escribir en la memoria local de un procesador?
8. Cómo viaja un dato entre dos procesadores? Los intermedios lo almacenan y luego lo forwardean hasta llegar al destino? O hay un camino establecido y el dato viaja directamente?
9. Los caminos son estáticos o dinámicos? Ya están establecidos por el diseñador del algoritmo o está permitida cierta flexibilidad en la elección de los caminos?
10. Se necesita un *handshake* entre los procesadores emisor y receptor? Esto es, el destino debe *reconocer* la recepción, o en el caso donde se establece primero un camino, emisor y receptor deben acordar que el camino se formó antes de comenzar el flujo de datos?
11. Los procesadores operan sincrónica o asincrónicamente?
12. Qué clase de procesador es usado por la red de interconexión? Qué clase de operaciones pueden ser realizadas por un procesador?

Todas estas preguntas son útiles, y por supuesto cada una tiene más de una respuesta. Sin embargo, algunas son más importantes que otras y sirven para distinguir claramente los submodelos, mientras otras sólo tratan con detalles.

### 2.5.1 Redes de Interconexión Dinámicas

Si se considera la implementación de una computadora de memoria compartida EREW-*PRAM* con $p$ procesadores y una memoria global de $m$ palabras, los procesadores están conectados a la memoria a través de un conjunto de elementos de intercambio (*switching*) que determinan la palabra de memoria que está siendo accedida por cada uno. En este modelo, cada procesador del ensamble puede acceder cualquiera de las palabras de memoria, siempre que una palabra no sea accedida por más de uno. Para asegurar esta conectividad, el número total de elementos de switching debe ser $\Theta(mp)$. Para una memoria

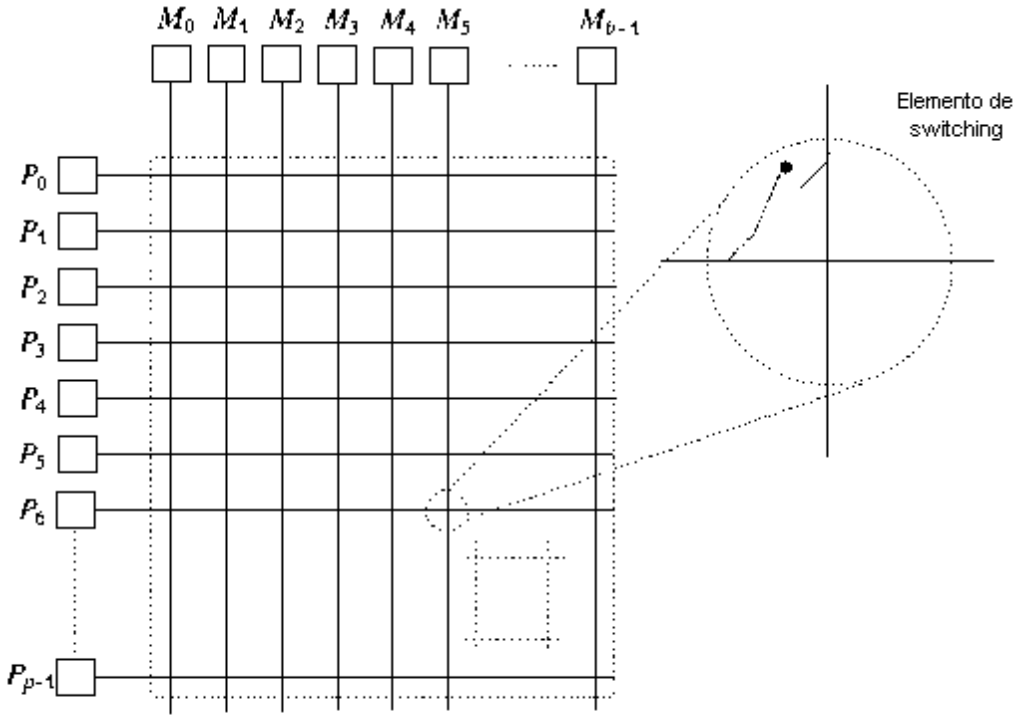


Figura 2.10: Crossbar switch completamente no bloqueante que conecta $p$ procesadores a $b$ bancos de memoria

de tamaño razonable, construir una red de switching de tal complejidad es demasiado costoso, lo que lo hace prácticamente imposible de realizar.

Una manera de disminuir la complejidad de la red de switching es reducir el valor de $m$ organizando la memoria en bancos. Un procesador switchea entre bancos y no entre palabras, reduciendo la cantidad de elementos entre los que se debe switchear. Esta es solo una aproximación débil al EREW-*PRAM*.

Algunas de las redes de interconexión dinámicas usadas en arquitecturas de memoria compartida prácticas son:

- *Crossbar Switching*. Emplea una grilla de elementos de switch. Es una red no bloqueante en el sentido de que la conexión de un procesador a un banco de memoria no bloquea la de otro procesador a otro banco (Figura 2.10). Cray Y-MP y Fujitsu VPP 500 son ejemplos de estas máquinas. Es escalable en términos de performance pero no de costo.

- *Basadas en Bus*. Los procesadores están conectados a memoria global por medio de un camino de datos común llamado *bus*, lo que constituye un sistema de fácil

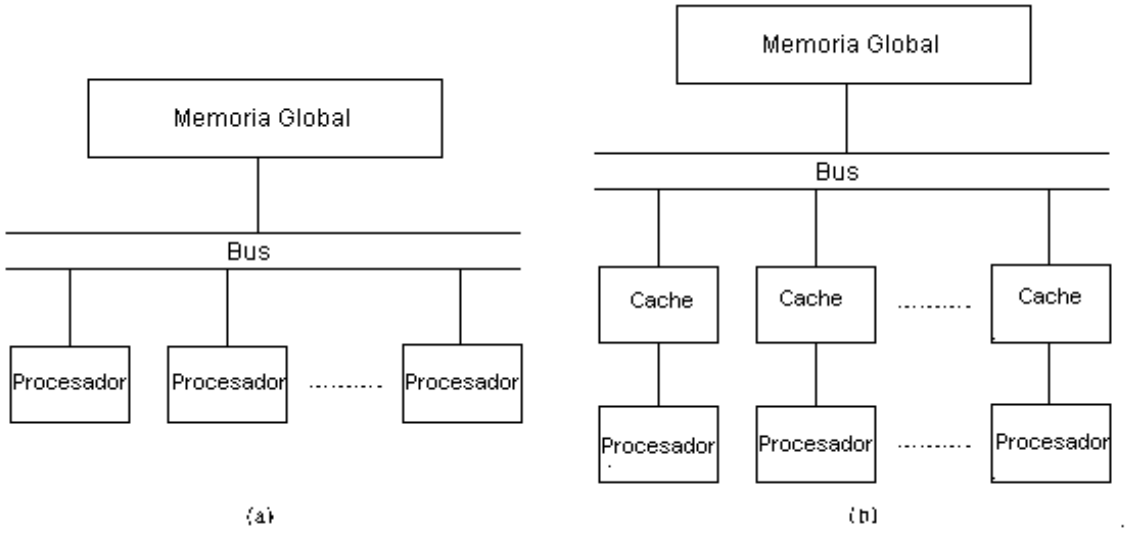


Figura 2.11: Arquitectura basada en bus (a) sin cache (b) con cache en cada procesador

construcción (Figura 2.11). Cada vez que un procesador accede a memoria global genera un pedido sobre el bus, y los datos viajan por éste, convirtiéndolo en un cuello de botella sobre todo al aumentar el número de procesadores. Una manera de aliviarlo es teniendo memoria cache local para reducir los accesos a memoria global. Es escalable en términos de costo pero no de performance.

- *Multistage*. Es una clase intermedia entre las dos anteriores (Figura 2.12). Es más escalable que el bus en términos de performance y más escalable que la crossbar en términos de costo. Consta de $p$ procesadores y $b$ bancos de memoria. Un ejemplo es la *omega network*.

### 2.5.2 Redes de Interconexión Estáticas

Las arquitecturas de pasaje de mensajes típicamente usan redes de interconexión estáticas. En este punto, uno de los aspectos más importantes trata con la *topología* de las redes; en las siguientes Secciones se detallan las más conocidas.

#### Red completamente conectada

Cada procesador tiene un link de comunicación directo a cada uno de los otros procesadores de la red (Figura 2.13(a)). Es ideal en el sentido de que pueden enviarse mensajes en un único paso. Son la contraparte estática de las redes crossbar switching, dado que en

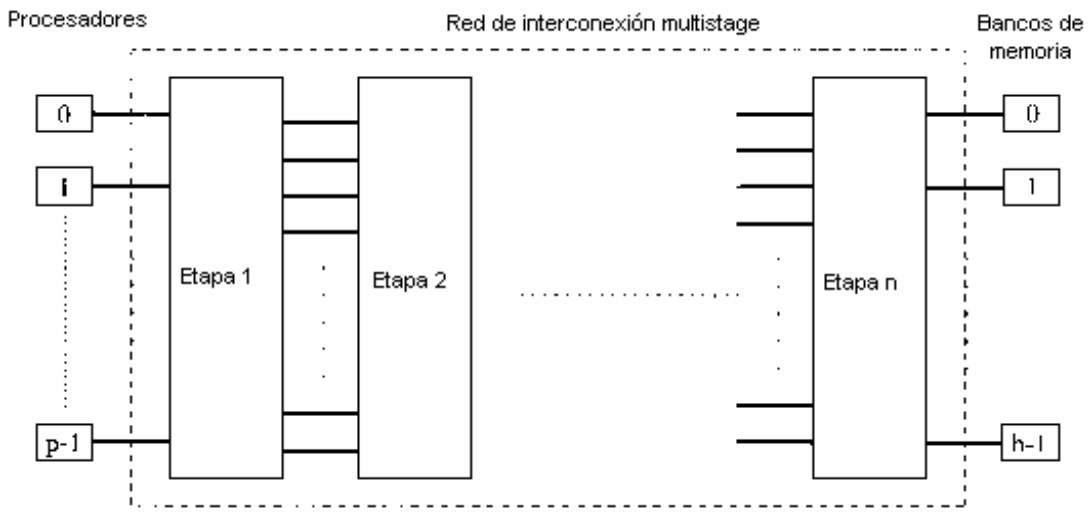


Figura 2.12: Esquema de red de interconexión multistage

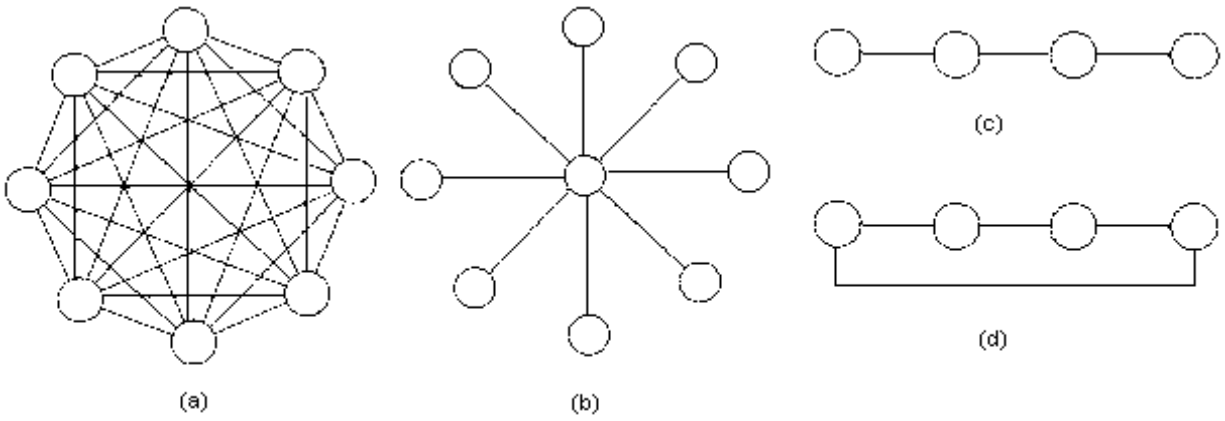


Figura 2.13: Topologías de red: (a) Red completamente conectada de 8 procesadores; (b) Conexión en estrella con 9 procesadores; (c) Arreglo lineal de 4 procesadores; (d) Anillo de 4 procesadores.

ambas la comunicación entre cualquier par de entrada/salida no bloquea la comunicación entre otro par. Sin embargo, las redes completamente conectadas pueden soportar comunicación sobre múltiples canales originadas en el mismo procesador, mientras los crossbar switches no. La mayor desventaja de esta clase es su costo.

### Estrella

Un procesador actúa como central, y el resto tiene un link de comunicación con él (Figura 2.13(b)). Es similar a la red basada en bus. Las comunicaciones entre cualquier par de procesadores es ruteada a través del central, así como el bus compartido forma el medio para toda comunicación en una red basada en bus. Por lo tanto, el procesador central es el cuello de botella en esta topología. Una red de estaciones de trabajo conectadas usando Ethernet puede usarse como máquina paralela: la Ethernet forma un medio común para transmitir mensajes entre procesadores. Tal red muestra características de performance similares a computadoras conectadas en estrella y basadas en bus.

### Arreglo lineal y Anillo

Una manera simple de conectar procesadores es en forma de arreglo lineal (unidimensional), haciendo que cada uno tenga un link directo a otros dos (excepto los extremos) (Figura 2.13(c)). Si se agrega un enlace que conecte los extremos ("*wraparound*") se obtiene un anillo (Figura 2.13(d)). Una forma de comunicar un mensaje es pasarlo repetidamente al procesador que se encuentra inmediatamente a la derecha (o la izquierda, de acuerdo a la dirección del camino más corto) hasta que llega a destino. Algunas máquinas paralelas que usan un anillo incluyen la ZMOB y CDC Cyberplus.

### Mesh

La *mesh bidimendional* es una extensión del arreglo lineal a dos dimensiones. Cada procesador tiene un link directo que lo conecta con otros 4 (Figura 2.14(a)). Si ambas dimensiones contienen igual número de procesadores entonces se habla de *mesh cuadrada*; en otro caso, se llama *mesh rectangular*. Con frecuencia, los procesadores de la periferia están conectados por *links wraparound*, obteniendo una *mesh wraparound* o *toro* (Figura 2.14(b)).

Un mensaje puede ser ruteado en la red enviándolo primero a lo largo de una de las dimensiones y luego de la otra hasta alcanzar su destino. Las extensiones comunes incluyen la *mesh tridimensional* (Figura 2.14 (c)) y la *mesh tridimensional wraparound*. Muchas máquinas comerciales se basan en esta topología. La DAP y la Paragon XP/S incluyen una mesh bidimensional, y la Cray T3D y J-Machine meshes tridimensionales.

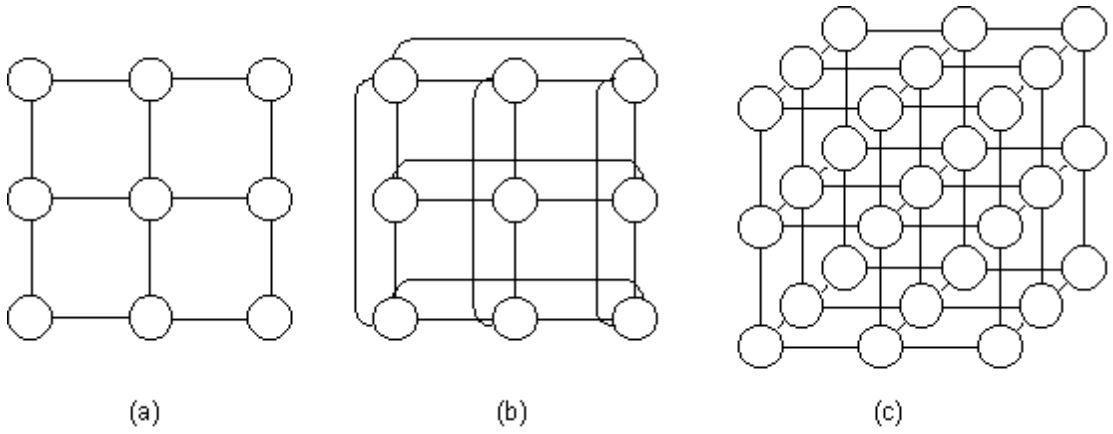


Figura 2.14: Meshes. (a) Bidimensional; (b) Bidimensional wraparound; (c) Tridimensional.

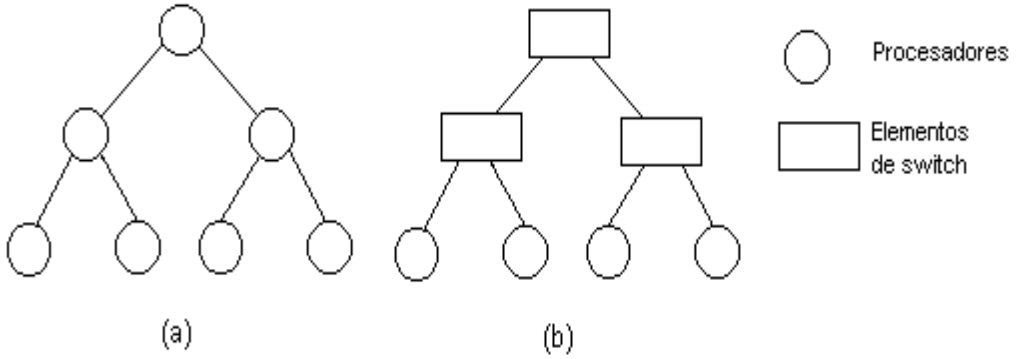


Figura 2.15: Arboles. (a) Binario completo; (b) Con elementos de switching.

### Arbol

En este caso existe sólo un camino entre cualquier par de procesadores (Figura 2.15(a)). Los arreglos lineales y las estrellas son casos especiales de árboles. Las redes en árbol estáticas tienen un procesador en cada nodo; existe una contraparte dinámica, donde los nodos en niveles intermedios son elementos de switching y los nodos hoja son procesadores (Figura 2.15(b)).

Para rutear un mensaje en un árbol, el nodo fuente envía el mensaje hacia arriba hasta que alcanza el procesador o el switch en la raíz del subárbol más chico que contiene al destino; luego el mensaje baja hasta ese procesador. Esta topología sufre un cuello de botella en los niveles superiores que puede aliviarse incrementando el número de links de comunicación entre procesadores cercanos a la raíz, obteniendo un *fat tree*. La máquina DADO usa un árbol binario estático, y la CM-5 se basa en una red fat tree dinámica.

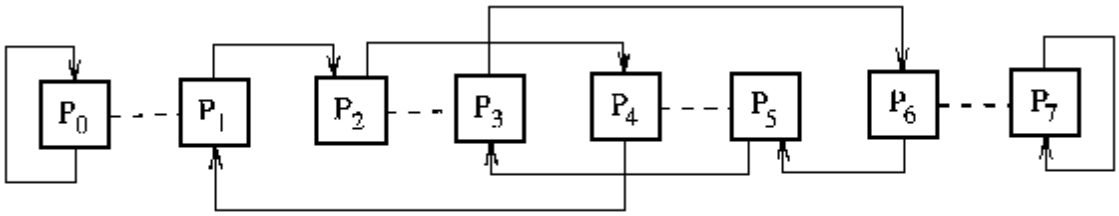


Figura 2.16: Red Shuffle Exchange

### Mesh de Arboles

$N$ procesadores se ubican en un arreglo de $\sqrt{N} \times \sqrt{N}$. Los procesadores en cada fila están interconectados formando un árbol binario, y de la misma forma los de cada columna. Las raíces de estos árboles binarios son los procesadores de la columna más a la izquierda y la fila de más arriba; las interconexiones de árbol son los únicos enlaces existentes.

### Pirámide

Una computadora paralela *pirámide unidimensional* se obtiene agregando links bidireccionales conectando procesadores en el mismo nivel en un árbol binario, formando un arreglo lineal en cada nivel. Este concepto puede extenderse a dimensiones mayores.

### Shuffle Exchange

Consta de $N$ procesadores ($N$ potencia de 2). En la interconexión *perfect shuffle*, una línea de comunicación conecta $P_i$ a $P_j$, donde $j = 2i$ para $i$ entre 0 y $N/2 - 1$ o $j = 2i + 1 - N$ para $i$ entre $N/2$ y $N - 1$ (Figura 2.16). De manera equivalente, la representación binaria de $j$ se obtiene por *shift* cíclico de $i$ una posición a la izquierda.

Si se revierten las direcciones de los links se obtiene la conexión *perfect unshuffle*. Si se tienen enlaces bidireccionales se denomina *red shuffle-unshuffle*. Si a esta última se le agregan enlaces bidireccionales que conectan los procesadores de número par con su sucesor (*links exchange*), se obtiene una *red shuffle-exchange*.

### Hipercubo

Un *hipercubo* es una mesh multidimensional con exactamente dos procesadores en cada dimensión. Un hipercubo $d$-dimensional consta de $p = 2^d$ procesadores (Figura 2.17).

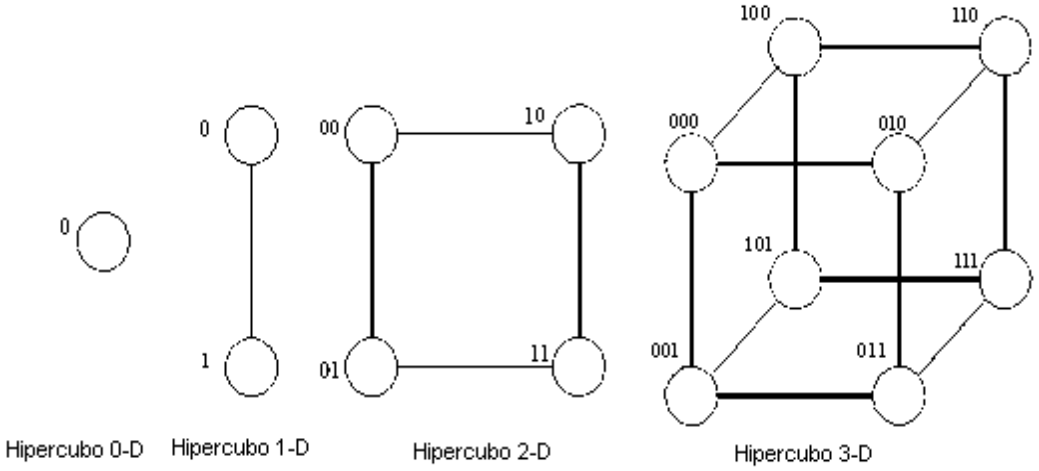


Figura 2.17: Hipercubo

Puede construirse recursivamente: un hipercubo cero-dimensional es un único procesador; el uni-dimensional se construye conectando dos cero-dimensionales; en general, un $(d+1)$-dimensional se construye conectando los procesadores correspondientes de dos $d$-dimensionales (uniendo los que tienen igual representación binaria y anteponiéndoles un 0 a uno y un 1 al otro). Los hipercubos tienen importantes propiedades, entre las que se encuentran:

1. Dos procesadores están conectados por un link directo si y sólo si la representación binaria de sus labels difiere en exactamente un bit.

2. En un hipercubo $d$-dimensional, cada procesador está directamente conectado a otros $d$.

3. Un hipercubo $d$-dimensional puede particionarse en dos $(d-1)$-dimensionales.

4. Los labels de procesadores en un $d$-dimensional contienen $d$ bits. Fijando $k$ bits cualquiera, los procesadores que difieren en las $d-k$ posiciones restantes forman un subcubo $(d-k)$-dimensional compuesto por $2^{d-k}$ procesadores. Dado que $k$ bits pueden fijarse de $2^k$ maneras distintas, hay $2^k$ subcubos de este tipo.

5. La cantidad de posiciones en que difieren los labels de dos procesadores se denomina *distancia de Hamming* entre ellos. El número de links de comunicación en el camino más corto entre dos procesadores es la distancia de Hamming entre sus labels.

Cosmic Cube, nCUBE-2 e iPSC son ejemplos de computadoras paralelas basadas en una red hipercubo.

**Redes d-cubo k-arios**

Un hipercubo $d$-dimensional, también llamado $d$-cubo, es una mesh $d$-dimensional con dos procesadores a lo largo de cada dimensión. En contraste, un anillo es una estructura unidimensional con $p$ procesadores a lo largo de su única dimensión. Estas topologías definen los extremos de una clase llamada *d-cubos k-arios*, donde $d$ es la *dimensión* de la red y $k$ es la *raíz*, que está definida como el número de procesadores a lo largo de cada dimensión.

### 2.5.3 Evaluación de redes de interconexión estáticas

Los criterios usados para comparar redes son variados, y pueden ayudar a determinar lo más adecuado para cada caso. La efectividad de la red difiere considerablemente de acuerdo a las aplicaciones y los entornos de operación, pero aún si estas variables son fijadas, deben elegirse métricas de performance y costo. Citando a Liszka *et al.*, determinar científicamente la mejor red es tan difícil como decir que un animal es mejor que otro [238]. Algunos de los items que pueden considerarse son:

- *Grado.* El grado de un procesador en una topología es el número de vecinos que posee. El grado de la red es el máximo de los grados de los procesadores. Es un criterio importante y debe considerarse cuidadosamente. Por un lado, un grado grande es interesante desde un punto de vista teórico, ya que muchos procesadores están "a un paso", pero un grado pequeño es preferible desde un punto de vista práctico.

- *Diámetro.* La *distancia* entre dos procesadores en una topología es el número de links en el camino más corto entre ambos. El *diámetro* de la red es la longitud de la mayor distancia entre todos los pares de procesadores. Dado que los procesadores necesitan comunicarse, y que el tiempo para que un mensaje viaje de uno a otro depende de la distancia que los separa, son preferibles las redes de diámetro chico. El diámetro de una red completamente conectada es 1, el de una estrella es 2, el de un anillo es $\lfloor p/2 \rfloor$, el de una mesh bidimensional es $2(\sqrt{p}-1)$, el de una mesh *wraparound* es $2\lfloor \sqrt{p}/2 \rfloor$, el de un hipercubo es $log\ p$.

- *Conectividad.* Es una medida de la multiplicidad de caminos entre dos procesadores cualquiera. Es deseable una red con alta conectividad, porque disminuye la contención por recursos de comunicación.

- *Ancho de Bisección y Ancho de Banda de Bisección.* El ancho de bisección es el mínimo número de enlaces que deben eleminarse para particionar la red en dos mitades. El ancho de bisección de un anillo es 2 (cualquier partición corta sólo dos links), el de una mesh con $p$ procesadores es $\sqrt{p}$ y una mesh con wraparound $2\sqrt{p}$,

el de un árbol y una estrella es 1, etc. El número de bits que pueden comunicarse simultáneamente por un link se denomina *ancho de canal*, y es igual al número de cables físicos en cada enlace; la velocidad pico a la cual un cable puede distribuir bits se llama *velocidad del canal*. La velocidad pico a la cual los datos pueden ser comunicados entre los extremos de un link se llama *ancho de banda del canal* (producto entre la velocidad y el ancho del canal). El *ancho de banda de bisección* se define como el mínimo volumen de comunicación permitido entre dos mitades de la red con un número igual de procesadores, y está dado por el producto del ancho de bisección y el ancho de banda del canal.

- *Longitud de los links.* Aunque los modelos son objetos abstractos, algunos pueden representar máquinas paralelas a ser implementadas. Por esto, una topología es más deseable que otra si es más eficiente, más conveniente y más extensible. Una red con longitud de links constante usualmente es más sencilla y eficiente para implementar.

- *Costo.* Pueden usarse distintos criterios para evaluar el costo de una red. Una forma es definir el costo en términos del número de links o de cables necesarios; otra es usar el ancho de banda de bisección (que brinda una cota inferior del área o el volumen en un packaging bidimensional o tridimensional respectivamente).

- *Latencia.* Cuando los datos se transfieren entre procesadores y memorias, las demoras en la red pueden causar tiempos ociosos y degradar la performance. El *multitasking* puede ayudar a limitarlo pero no puede mejorar el tiempo de una única tarea.

- *Tolerancia a fallas.* El entorno de la red influye en la importancia de este punto (por ejemplo, un satélite inaccesible versus un laboratorio). Para cuantificar métricas tales como el número de fallas toleradas y la degradación que resulta de una falla, se debe establecer un modelo de falla y un criterio de tolerancia común.

- *Capacidad de permutación.* Una permutación es un conjunto de pares fuente-destino que pueden representarse matemáticamente como una biyección del conjunto $\{0, 1, \cdots, N-1\}$ en sí mismo. En este contexto, una permutación es la transferencia de un item de datos desde cada procesador a otro único, con todos transmitiendo simultáneamente. Distintas redes pueden requerir diferentes cantidades de tiempo para procesar varias permutaciones; Pueden ocurrir en máquinas MIMD, cuando una comunicación está precedida por una barrera de sincronización, y en SIMD.

- *Particionabilidad.* Algunas redes pueden particionarse en subredes independientes, cada una con las mismas propiedades que la original. Esto permite múltiples usuarios, ayuda en la tolerancia a fallas, soporta paralelismo de subtareas y provee el subconjunto de procesadores de tamaño óptimo para una tarea dada.

- *Efectividad de costo.* Es la performance de la red dividida por el costo de implementarla. Si se usa como medida científica, debería definirse en términos del hardware

necesario, no de costo monetario.

- *Otros.* Incluyen *throughput* (número de mensajes que llegan a destino durante un intervalo de tiempo), *tiempo de multicasting* (cúanto le lleva a un procesador enviar el mismo mensaje a múltiples procesadores), *escalabilidad* (rango de tamaños de máquina para los cuales un diseño de red es apropiado), etc.

Una dificultad al comparar dos diseños de red es determinar si son de igual costo, para que la comparación sea justa. Por ejemplo, para más de 16 procesadores, ¿el número de enlaces y complejidad de switcheo para un hipercubo es mayor que para una mesh? Para comparar la performance de las dos redes, ¿cómo debería aumentarse la mesh para hacerlas de igual costo?

El problema de realizar comparaciones justas son los numerosos parámetros de diseño y uso que pueden variar. Los diseñadores e implementadores deben entender cómo estos parámetros afectan la performance de la red. El tema es tratado ampliamente en [238].

### 2.5.4 Mecanismos de ruteo para Redes Estáticas

Los algoritmos eficientes para rutear un mensaje a su destino son críticos para la performance de las máquinas paralelas. Un *mecanismo de ruteo* determina el camino que sigue un mensaje a través de la red para ir desde el procesador fuente al destino. Toma como entrada los procesadores fuente y destino de un mensaje, puede usar información sobre el estado de la red, y retorna uno o más caminos.

Los mecanismos de ruteo pueden clasificarse como *mínimos* (siempre elige uno de los caminos más cortos, pero puede llevar a congestión) o *no mínimos* (eventualmente rutea por un camino más largo pero evita congestión).

También pueden ser clasificados por cómo usan información respecto del estado de la red. Un *ruteo determinístico* obtiene un camino único para un mensaje basado en su fuente y destino, sin tener en cuenta el estado (pueden hacer uso desigual de los recursos de comunicación). Por el contrario, un *ruteo adaptivo* usa información de estado para determinar el camino del mensaje (detecta congestión y la evita).

#### Costos de Comunicación en Redes de Interconexión Estáticas

El tiempo usado en comunicar información de un procesador a otro con frecuencia es una gran fuente de overhead al ejecutar programas en una máquina paralela. El tiempo para comunicar un mensaje entre dos procesadores de la red se denomina *latencia de comunicación*, y es la suma del tiempo para preparar el mensaje para su transmisión y el tiempo para atravesar la red. Los parámetros principales que determinan la latencia son:

- *Tiempo de startup* ($t_s$). Es el requerido para manejar un mensaje en el procesador que envía. Incluye los tiempos para preparar el mensaje (agregar encabezado, cola e información de corrección de errores), ejecutar el algoritmo de ruteo, y establecer una interfase entre el procesador local y el router. En esta demora se incurre sólo una vez para una transferencia de mensaje simple.

- *Tiempo per-hop* ($t_h$). Cuando un mensaje deja un procesador, toma una cantidad de tiempo alcanzar el próximo procesador en su camino. El tiempo que le toma al encabezado de un mensaje viajar entre dos procesadores conectados directamente es $t_h$.

- *Tiempo de transferencia por palabra* ($t_w$). Si el ancho de banda del canal es de $r$ palabras por segundo, entonces cada palabra toma $t_w = 1/r$ para atravesar el link. Este es el tiempo $t_w$.

Muchos factores influyen en la latencia de comunicación de una red, tales como la topología y las técnicas de switcheo. Entre estas últimas las dos más usadas son los ruteos *store-and-forward* y *cut-through* [209].

## 2.6 Algunas arquitecturas reales

En esta Sección se describen algunos casos de arquitecturas de máquinas paralelas. No se intenta cubrir todo el espectro sino sólo brindar ejemplos reales.

### 2.6.1 Connection Machine

Las Connection Machines fueron construidas por Thinking Machines Corporation. La CM-1 es una máquina SIMD; los procesadores están organizados en un hipercubo 12-dimensional con 16 elementos de procesamiento en cada vértice.

La CM-2 incorpora 65536 procesadores de 1 bit (celdas de procesamiento), cada uno con su propia memoria. Cada unidad de procesamiento (UP) contiene 4096 bits de memoria. Hay 16 procesadores en un chip, y 32 chips en cada una de las 128 placas de circuito impreso. Los 16 procesadores en un chip están enlazados por un switch que permite una conexión directa entre cualquier par de UPs. Cada dispositivo de ruteo de cada chip está conectado a otros 12. La comunicación entre procesadores puede realizarse por muchas rutas, facilitando la transmisión de datos aún si una ruta está ocupada. La velocidad promedio para la mayoría de las aplicaciones es de 2 billones de operaciones por segundo.

La CM-5 consta de nodos de procesamiento, control y entrada/salida y una interfase conectados por dos redes escalables que manejan la comunicación de información de

datos y control [166]. Cada nodo de procesamiento tiene un RISC (Reduced Instruction Set Computer) de 64 bits, 32 MB de memoria e interfase a las redes de datos y control. La CM-5 puede tener hasta 16384 procesadores, aunque la mayoría de las implementaciones cuenta con muchos menos (hasta 1024). La memoria puede llegar hasta 512 GB. Entre las principales características se encuentran la escalabilidad, la memoria distribuida y direccionamiento global, y la ejecución distribuida y sincronización global. Es MIMD aunque con aspectos encontrados sólo en SIMD. Los procesadores pueden operar independientemente cuando no es esencial la sincronización de instrucciones. Se usa una topología fat-tree en lugar del hipercubo de CM-2. Soporta un sistema operativo basado en Unix.

Las Connection Machine utilizan una computadora host front-end (en general del tipo *Sun*). El host dialoga con las celdas de procesamiento a través de un microcontrolador que actúa como amplificador de ancho de banda. Dado que las celdas pueden ejecutar instrucciones más rápido que lo que puede dictar el host, éste especifica *macroinstrucciones* de más alto nivel que son interpretadas por el microcontrolador para producir las *nanoinstrucciones* que llegan a las celdas. Entre los lenguajes utilizados se encuentran LISP y C extendidos. Las Connection Machines se aplicaron en áreas de investigación para el gobierno y en comercio internacional así como en Universidades con propósitos educativos.

### 2.6.2 MasPar Machines

MasPar es una compañía formada en 1988 por DEC (Digital Equipment Corporation), y representa Massively Parallel Machine. Produce máquinas SIMD de las series MP. El concepto de masivamente paralelo es una máquina que incorpora cantidades masivas de elementos de procesamiento. Usando una arquitectura de memoria distribuida, que combina memoria local y procesador en cada nivel de la red de interconexión, se pueden crear máquinas con un número casi infinito de procesadores.

La arquitectura de la MP-1 consta de elementos de procesamiento (PEs) conectados como mesh bidimensional. La máquina está manejada por un front-end (típicamente *VAX*). Pueden conectarse dispositivos de entrada/salida de alta velocidad, y es posible el acceso directo al bus de memoria DEC. Los PEs son diseñados especialmente por MasPar: son RISC agrupados en clusters de 16 por chip. Cada cluster tiene las memorias y conexiones del PE a la red de comunicaciones. Las instrucciones son manejadas por la ACU (Array Control Unit), procesador RISC standard de Texas Instruments. La topología en forma de grilla permite la comunicación con 8 vecinos. El sistema operativo es un Unix front-end. Los lenguajes soportados son ANSI C y Fortran de MasPar (versión de Fortran 90). La performance es de 1.2 GFlops para una máquina con 16384 PE.

La MP-2 se encuentra en tamaños de 1024, 2048, 4096, 8192 y 16384 procesadores en su arreglo de PEs; éste es controlado por una ACU y tiene su propia memoria para datos

e instrucciones. Cada PE recibe la misma instrucción desde la ACU simultáneamente, y sólo aquellos elementos del conjunto activo (que puede definir el usuario) llevan a cabo las instrucciones. Tanto la ACU como el arreglo de PEs están contenidos en la DPU (Data Parallel Unit), donde se realiza todo el procesamiento paralelo.

### 2.6.3 Máquinas Cray

Los sistemas supercomputador Cray están diseñados para manejar las necesidades de alta performance en investigación científica, manufactura y gobierno. La primera fue la Cray-1 (1976), instalada en Los Alamos National Laboratory, con una performance pico de 133 Mflops [80].

La Cray SV1 es la cuarta generación de sistema vectorial CMOS. La Cray MTA combina arquitectura multithreaded con memoria compartida uniforme escalable y un entorno de programación paralelo fácil de usar. La Cray T3E es adecuada para problemas altamente paralelos, y escala desde cientos a miles de procesadores. El sistema Cray SV2 es el primero en ofrecer capacidades vectoriales y de procesamiento masivamente paralelo en una arquitectura única. Los sistemas Cray soportan la mayoría de aplicaciones, cientos de usuarios y procesos, petabytes de datos y requerimentos críticos.

La Cray T3E es una supercomputadora MIMD (con soporte SIMD) altamente escalable y con el record mundial de performance sostenida para una aplicación de 64 bits. Es usado para cumplir con desafíos científicos y técnicos en aplicaciones que incluyen electromagnetismo, química, dinámica de fluidos, predicción del clima y procesamiento sísmico tridimensional. La performance escalable se deriva combinando microprocesadores y comunicaciones interprocesador ultrarrápidos con sistema operativo escalable y capacidades de entrada/salida de alta performance. Los procesadores están fuertemente acoplados por un toro tridimensional bidireccional con extremadamente baja latencia y alto ancho de banda. La entrada/salida se realiza por múltiples puertos en uno o más canales GigaRing, accesibles y controlables desde todos los PEs. Hay 8 PEs por módulo, y de 40 a 2176 PEs por sistema (en incrementos de 8). La memoria (cache coherent, físicamente distribuida y globalmente direccionable) va desde 10 GB a 1TB.

La arquitectura multithread de Cray MTA ofrece memoria compartida uniforme escalable, liberando al programador de la distribución de datos. Entre las características se encuentran: datos, direcciones e instrucciones de 64 bits, hasta 128 threads por procesador (cada thread es un flujo con su propio contador de instrucciones, conjunto de registros y palabra de estado), hasta 8 referencias a memoria concurrentes por thread, aritmética de punto flotante, extensiones de Fortran y C, paralelización y vectorización automática, entrada/salida transparente y escalable, procesador computacional. Cada placa está conectada a la red por cuatro nodos de ruteo, lo que permite transferencias de datos desde y hacia memoria a alta velocidad en ambas direcciones. Los nodos están conectados en

estructura de grupo de Cayley, brindando menor latencia que una topología bi o tridimensional. MTA utiliza tecnología CMOS, que permite funcionar con menor cantidad de partes, consumir menos y disminuir las conexiones, mejorando la confiabilidad, performance y costo.

La Cray SV1 fue elegida la mejor supercomputadora del 2001 por los lectores del Scientific Computing and Instrumentation Magazine, en base a "su valor por precio, soporte técnico, facilidad de uso, calidad y confiabilidad". Mantiene la tradición de escalabilidad y agrega la capacidad de ejecutar computaciones vectoriales en memoria cache vectorial de alto ancho de banda, lo que alivia la contención de memoria principal y acelera los cálculos en vectores. Provee una arquitectura innovadora: el Procesador Multi-Streaming, que combina 4 procesadores de 2 GFlops para crear un pipe 8-vector con una performance pico de 8 Gflops. En un cluster, escala hasta 32 nodos con más de 1 Teraflop de capacidad pico de CPU y más de 1 Terabyte de memoria.

La SV2 brinda ancho de banda de memoria, interconexiones y capacidades de procesamiento vectorial excepcionales, y escala hasta varias decenas de Teraflops.

### 2.6.4 IBM SP2

Es un sistema paralelo de propósito general con procesadores RISC 6000 diseñado para tratar un amplio número de aplicaciones [183]. Se basa en una arquitectura escalable de pasaje de mensajes con memoria distribuida. Varía entre 2 y 512 PEs. Los nodos están conectados por una red multistage de alta performance, y cada uno tiene su copia de sistema operativo AIX.

Para hacerla escalable a un gran número de procesadores, la memoria está distribuida, con una parte cercana a cada procesador para permitir rápido acceso local. Típicamente hay una imagen del micro-kernel de sistema operativo en cada nodo, pero no son independientes ya que están fuertemente conectadas al menos a nivel de manejador de memoria virtual para presentar un espacio de direcciones global único. Puede haber múltiples trabajos paralelos corriendo en la máquina simultáneamente, cada uno controlando un conjunto disjunto de nodos. Dada la topología de la red de conexión, no hay restricciones sobre el tamaño de la partición paralela o la ubicación topológica de los nodos, lo que provee flexibilidad en el diseño de aplicaciones y el manejo de jobs y recursos.

Provee un subsistema de entrada/salida que escala en performance y capacidad con el poder de cómputo del sistema. Los nodos RISC 6000 POWER2 superescalares ejecutan múltiples instrucciones concurrentemente cada ciclo (en lugar de superpipelined corriendo el procesador en muy alta frecuencia). La unidad de instrucción puede decodificar y ejecutar varias instrucciones en cada ciclo para mantener todas las unidades ocupadas. Esto se aparea con una jerarquía de memoria de alta performance y gran capacidad. Los nodos están interconectados por un switch diseñado para obtener escalabilidad, baja

latencia, alto ancho de banda, bajo overhead de procesador y comunicación flexible y confiable. La topología del switch es una red packetswitched any-to-any multistage o indirecta, similar a la Omega, lo que permite escalar el ancho de banda de bisección linealmente con el tamaño del sistema.

El entorno paralelo tiene 4 componentes principales: *Message-Passing Library* (MPL) que es una librería de comunicaciones avanzada y soporta pasaje de mensajes explícito para programas C o Fortran; *Parallel Operating Environment* (POE) que brinda un entorno de usario para iniciar, monitorear y controlar la ejecución de una aplicación paralela; *Visualization Tool* (VT), que permite monitorear y trazar la visualización de una aplicación paralela; y *Parallel Debugger* (PD), que es un *debugger* a nivel de código con una línea de comandos y una interfase gráfica.

Dado que los sistemas de archivos distribuidos de Unix (NSF, ASF, DFS) no satisfacen el ancho de banda extremadamente alto en el acceso a archivos de datos requeridos por aplicaciones intensivas, SP2 creó el PFS (Parallel File System), que soporta acceso paralelo (desde una aplicación corriendo en paralelo en múltiples nodos) a un archivo particionado en varios nodos.

### 2.6.5 SGI Origin 2000

Es un multiprocesador escalable, de espacio de direcciones compartidas *cache-coherent*, con acceso no uniforme a memoria (ccNUMA) [304, 305], y permite el rápido crecimiento de la plataforma. Los recursos del sistema pueden ser desplegados como arreglos fuertemente integrados de pequeños sistemas de la serie SGI 2000 o como una única gran máquina de memoria compartida. Puede escalar desde un procesador hasta 512.

Los multiprocesadores sincrónicos típicos se basan en un bus y tienen un ancho de banda fijo que colapsa al agregar recursos de memoria y procesadores. La arquitectura ccNUMA no se basa en un bus, sino que usa una serie de *switches crossbar* no bloqueantes como estructura de interconexión, agregando entrada/salida incremental cuando crece la configuración del sistema.

En la Origin 2000, un número de nodos de procesamiento están unidos por una estructura de interconexión llamada NUMAlink. Cada nodo de procesamiento (*node board*) contiene procesadores, una porción de memoria compartida, un directorio para coherencia de cache, y dos interfases: una que la conecta con otros dispositivos de entrada/salida en el sistema y otra que lo vincula con *node boards* adicionales.

La estructura de interconexión vincula los *node boards*, pero difiere de un bus en varios aspectos. Un bus puede ser usado por un solo procesador a la vez. La estructura es una mesh de transacciones múltiples, simultáneas y dinámicamente alocables (las conexiones se realizan entre procesadores según se necesite).

El building-block más chico de ccNUMA es el *node board*, que contiene dos MIPS RISC R10000 de 64 bits, con cache y memoria asociada, y está conectado a un router. El *crossbar switch* de alta performance (Hub, 1.6 GB por segundo), diseñado por SGI, provee ancho de banda procesador a memoria al agrega módulos. El Hub también controla la interfase a todos los procesadores, memoria y otros dispositivos de entrada/salida, brindando la infraestructura necesaria para construir un sistema de 128 procesadores donde cada uno puede acceder a toda la memoria (hasta 256 GB) y cada slot de E/S (192 en total).

### 2.6.6 Transputers

El transputer es un microprocesador RISC de alta performance con memoria on-chip y links seriales usados para comunicación interproceso entre nodos [185, 261]. Un único transputer no representa una arquitectura paralela distribuida, pero brinda un *building block* para construir sistemas paralelos conectándolos en redes a través de los links. Sus características de diseño son simplicidad, regularidad, alto grado de escalabilidad, alta performance y alta conectividad.

Un transputer consta de procesador, memoria, y links de entrada salida, todo conectado a un bus de 32 bits. El bus también comunica con la interfase de memoria externa, permitiendo usar memoria local adicional. Puede aumentarse las instrucciones del RISC con un conjunto de instrucciones extendido específico de la aplicación. Las instrucciones están implementadas en microcódigo en lugar de ser *hardwired* como en la mayoría de los RISC, convirtiendo al transputer en un PE flexible que puede adaptarse a distintas aplicaciones.

Pueden ser programados en la mayoría de los lenguajes de alto nivel, y están diseñados para asegurar eficiencia. Para aprovechar los beneficios de la arquitetura puede usarse Occam [184], que provee las ventajas de un lenguaje de alto nivel, máxima eficiencia y capacidad de usar las características especiales. Occam brinda un framework para diseñar sistemas concurrentes de la misma forma en que lo hace el álgebra booleana para los sistemas electrónicos a partir de puertas lógicas.

El diseño del procesador del transputer explota la disponibilidad de memoria on-chip rápida teniendo sólo un pequeño número de registros (6 en la CPU). El scheduler (en microcódigo) provee dos niveles de prioridad y permite cualquier número de procesos concurrentes ejecutando juntos, compartiendo el tiempo de procesador. La comunicación entre procesos es mediante canales; en Occam es punto a punto, sincronizada y sin buffer. Cuando los procesos se encuentran en distintos transputers se utilizan los links externos para realizar la comunicación.

### 2.6.7 Clusters

El bajo costo relativo de las computadoras personales y sus contínuas mejoras de performance ha llevado a utilizarlas, conectándolas en una red, como máquinas paralelas, y de esta manera se han convertido en una de las principales infraestructuras de cómputo para Ciencia e Ingeniería [16].

En [272], Pfister define un *cluster* como un tipo de sistema paralelo o distribuido que consta de un conjunto de computadoras completas interconectadas y se usa como un único recurso de cómputo unificado. Aunque no es una definición aceptada universalmente, da una buena idea del concepto.

El término *computadora completa* indica un nodo que comprende uno o varios procesadores, una cache, una memoria principal, un disco, una interfase de entrada/salida, y un sistema operativo propio. De esta manera, puede ser una computadora personal, una workstation o un multiprocesador sincrónico. La red de interconexión usa tecnologías standard como Ethernet o de alta performance como Myrinet. El número de nodos varía desde dos hasta unos pocos miles.

El término *único recurso de cómputo unificado* se refiere a un sistema con algunas funcionalidades de SSI (*single system image*) como un solo punto de entrada (uno se puede conectar o *loggear* al cluster en lugar de a un nodo particular), sistema de archivos único, colas de trabajo coordinadas y transparencia de locación, e imagen de sistema única para administración. Típicamente estas carcterísticas son provistas por una capa middleware sobre el sistema operativo o por hardware [228].

Los clusters tienen varias ventajas sobre las arquitecturas paralelas más convencionales; la ms importante es su precio, que es comparativamente bajo porque utilizan componentes existentes. Otras son la escalabilidad y el hecho de que los usuarios con frecuencia pueden mantener su entorno usual. La distinción entre clusters y computadoras paralelas de memoria distribuida es gradual: depende principalmente del grado al cual se optimiza y se integra al sistema la red de comunicaciones.

Las arquitecturas de cluster varían ampliamente. Primero, los *clusters dedicados* están diseñados para usarlos como máquinas paralelas. Con frecuencia se encuentran agrupados de manera compacta, esto es, ubicados en un área física chica y no contienen periféricos tales como teclado y monitor. Algunos son construidos, vendidos y soportados técnicamente por vendedores de productos, mientras otros son compuestos por el usuario.

Los *clusters campus-wide* conectan computadoras personales o workstations utilizadas normalmente por un solo usuario, e intentan utilizar recursos ociosos en una red y aprovecharlos en trabajo útil. Obviamente son más lentos que los dedicados, y se usan en aplicaciones de grano más grueso. Además de la velocidad, otra diferencia es que la red está expuesta, esto es, tanto el tráfico relacionado con el cluster como el no relacionado viajan por las mismas conexiones; en consecuencia, hay un tema importante de seguri-

dad. Por otro lado, los *clusters campus-wide* son generalmente heterogéneos, mientras los dedicados son homogéneos. Entre ambos extremos, existen numerosas formas intermedias.

Se han identificado tres oportunidades para las cuales los clusters benefician a los usuarios: mejora de performance del sistema de archivos y de memoria virtual usando la DRAM agregada como una cache gigante para disco; almacenamiento más barato, de alta disponibilidad y escalable usando arreglos redundantes de discos; y finalmente, múltiples CPUs para cómputo paralelo y distribuido.

El desafío es brindar al usuario el tiempo de respuesta rápido y predecible de una máquina dedicada mientras se permite que tareas que son demasiado grandes utilicen recursos de la red. La performance cruda de la red brinda parte de la solución, pero debe ponerse especial atención a la memoria como un recurso interactivo y a las suposiciones de *scheduling* de los programas paralelos. Al examinar características de uso típicas de *workstations* dedicadas y de MPPs dedicadas se ve que los dos tipos de cargas de trabajo pueden ser combinadas de maneras complementarias, dando la capacidad de detectar recursos ociosos y migrar procesos. Esto ultimo depende críticamente de una red rápida y un sistema de archivos paralelo construido fuera de los discos de las workstations de esa red.

Dentro de los clusters de alta performance típicos se encuentran Beowulf, HPVM y ASCI Blue Mountain. Los Beowulf son una clase de los dedicados en los cuales los usuarios compran y ensamblan las componentes por sí mismos [36]. El número de nodos varía de menos de diez a unos pocos cientos. Típicamente los nodos corren sistema operativo Linux.

HPVM significa High Performance Virtual Machine, y es un proyecto de investigación universitario. Un ejemplo de este sistema el el cluster HPVM III, que comprende alrededor de 100 nodos, donde cada nodo es un multiprocesador sincrónico de dos procesadores con sistema operativo Windows NT, usa tecnología Fast Ethernet y Myrinet para el tráfico interno y puede ser accedido desde el exterior a través de TCP/IP.

El ASCI Blue Mountain es un sistema de Silicon Graphics, con buen ranking en la lista de Top 500 supercomputadores. Comprende alrededor de 50 nodos, cada uno de los cuales es internamente un SGI Origin de 128 procesadores.

# Capítulo 3

# Paradigmas Paralelos. Diseño de Algoritmos

## 3.1 Introducción

Visto desde alto nivel, muchos algoritmos paralelos son puramente secuenciales, con la misma estructura global que un algoritmo diseñado para una computadora serial standard. Esto es, puede haber una fase de entrada e inicialización, luego una computacional y finalmente una de salida y terminación. Sin embargo, las diferencias se ponen de manifiesto dentro de cada fase. Por ejemplo, durante la fase computacional, un algoritmo paralelo eficiente puede ser inherentemente distinto de su contraparte secuencial.

Para cada una de las fases de una computación paralela, con frecuencia es útil pensar que se opera simultáneamente sobre una estructura de datos completa; este es el enfoque estilo SIMD, pero las operaciones pueden resultar complejas. Por ejemplo, intentar actualizar todas las entradas de una matriz, árbol o base de datos, viéndolo como una operación única y compleja. Para una máquina de grano fino, ésto podría implementarse teniendo un único ítem de datos (o pocos) por procesador, y luego usar un algoritmo puramente paralelo para la operación.

Para máquinas de grano medio o grueso, operar sobre estructuras enteras puede implementarse mezclando enfoques seriales y paralelos. En tales arquitecturas, cada procesador usa un algoritmo secuencial eficiente aplicado a la porción de los datos que contiene, y se comunica con otros procesadores de la arquitectura para intercambiar datos críticos. Para maximizar la eficiencia, puede hacerse que cada procesador envíe los datos necesarios a sus vecinos, luego realice la operación sobre sus datos locales que no dependen de datos de otros, y finalmente lo haga sobre los elementos que dependen de los vecinos. Desafortunadamente, mientras esto minimiza el posible overlap entre comunicación y cálculo, también complica el programa pues debe reorganizarse el orden de las computaciones

dentro de un procesador.

Un *paradigma de programación* es *una clase de algoritmos que resuelve problemas distintos, pero que tienen la misma estructura de control* [52]. Para cada paradigma puede escribirse un programa general que define la estructura de control común; éste es llamado *esqueleto algorítmico*, *programa genérico* o *"template" de programa* [69].

Dentro de la programación se encuentran una diversidad de paradigmas: estructurada, funcional, lógica, imperativa, orientada a objetos, concurrente, paralela, etc. En el caso particular de la programación paralela, pueden distinguirse una serie de paradigmas que permiten encuadrar los problemas en alguno de ellos y simplifican la tarea del programador, ya que se puede generar una cantidad de procesos pero basta con programar una sola copia. Dentro de cada paradigma, los patrones de comunicaciones son prácticamente siempre los mismos.

Dado que un modelo es sólo una máquina abstracta, existen modelos a diferentes niveles de abstracción. Por ejemplo, cada lenguaje de programación es un modelo en algún sentido, pues brinda una visión simplificada del hardware subyacente. Esto hace difícil la comparación por causa de los distintos niveles de abstracción.

Al momento de resolver un problema usando un programa paralelo se tienen distintas posibilidades. Una es utilizar *programas preescritos*: existen códigos paralelos para bases de datos paralelas, algoritmos genéticos, álgebra lineal, etc; otra opción es sacar provecho de las *librerías paralelas*, lo cual es más rápido y robusto que la tercera posibilidad: *escribir el código desde cero* (la más dura pero también más flexible, adaptable a las distintas necesidades, y la que potencialmente debiera permitir una mayor eficiencia).

Es importante notar que no todos los problemas son paralelizables. Por ejemplo, si se intenta calcular la serie de Fibonacci $(1, 1, 2, 3, 5, 8, 13, 21, \cdots)$ utilizando la fórmula $F(k+2) = F(k+1) + F(k)$, el resultado de cada paso $F(k+2)$ requiere el resultado de los dos pasos previos, por lo que los tres ítems no pueden ser calculados simultáneamente. Si el problema es paralelizable, entonces debe diseñarse el algoritmo que lo resuelva, ajustándolo a alguno de los diferentes tipos de paralelismo (perfecto, de datos, de control, etc.).

Con respecto a los tipos de aplicaciones, puede distinguirse entre las *regulares* (aquellas en que las estructuras de datos son arreglos densos y los accesos a estas estructuras pueden ser bien caracterizados en tiempo de compilación) y las *irregulares* (en las que algunas de las estructuras de datos usadas pueden ser determinadas sólo en tiempo de ejecución) [282].

## 3.2 Tipos de paralelismo

### 3.2.1 Paralelismo perfecto

También se conoce como *embarrasingly parallel*, donde las mejoras de *speedup* pueden ser considerablemente buenas. La aplicación (que es paralela *por naturaleza*) es dividida en conjuntos de procesos que requieren poca o nula comunicación. Es la mejor y más simple clase de paralelismo cuando es aplicable.

Un ejemplo son las técnicas de Monte Carlo, consistentes en tomar un punto de partida al azar, encontrar una solución, guardar el resultado, y repetir hasta que la solución encontrada sea "suficientemente buena". Cada procesador puede realizar esto sin necesidad de comunicarse con los otros. Son algoritmos probabilisticos para problemas que no pueden ser resueltos de manera exacta. Se utiliza para definir distribuciones de probabilidad para estructuras atómicas.

Las computaciones evolutivas son buenas candidatas para esta clase de paralelismo: la idea de subpoblaciones (o *modelo de islas*) implica la evolución paralela de diferentes poblaciones con intercambios relativamente infrecuentes entre ellas.

Los clusters de workstations representan un tipo de arquitectura naturalmente adecuada para este tipo de aplicaciones, dado que generalmente proveen buena potencia de cómputo en cada máquina aunque con baja performance de comunicación.

### 3.2.2 Paralelismo de datos

En algunos problemas, varios ítems de datos están sujetos a procesamiento idéntico. Entonces, pueden ser paralelizados asignando elementos a varios procesadores, cada uno de los cuales realiza las mismas computaciones sobre sus datos. El *paralelismo de datos* (o *descomposición de dominio*) está caracterizado por la ejecución paralela de la misma operación sobre diferentes partes de un gran conjunto de datos. Las áreas del dominio son mapeadas a distintos PEs; por ejemplo, matrices distribuidas, búsquedas en diferentes partes de una base de datos a partir de una consulta, modelización de elementos finitos en aviones (con distintas partes del avión en cada procesador), etc.

Este tipo de paralelismo influyó fuertemente en el desarrollo de una clase de lenguajes de programación (*lenguajes de programación data-parallel*). Los programas escritos en tales lenguajes son llamados *programas data-parallel*; de todas formas, puede expresarse este paradigma en lenguajes que no son data-parallel.

Un programa data-parallel contiene una única secuencia de instrucciones, cada una de las cuales se aplica a los elementos de datos en forma *lock-step*. La operación paralela

puede ser elemental o compleja, y de acuerdo a esto, se distinguen dos estilos básicos de paralelismo de datos: SIMD y SPMD. En el primer caso, una unidad de control global hace broadcast de las instrucciones a los procesadores (los cuales contienen los datos), y éstos ejecutan la corriente de instrucciones sincrónicamente.

Los programas data-parallel también pueden ser ejecutados en máquinas MIMD, pero en este caso la ejecución estricta resulta en código ineficiente pues requiere sincronización global después de cada instrucción. Una solución es *relajar* la ejecución sincrónica de instrucciones; en dicho modelo, denominado SPMD (*single program, multiple data*), cada procesador ejecuta el mismo programa asincrónicamente, y la sincronización tiene lugar sólo cuando los procesadores necesitan intercambiar datos. De esta forma, el paralelismo de datos puede explotarse en una computadora MIMD aún sin usar un lenguaje de programación data-parallel explícito.

Otro criterio de clasificación es la estructura del conjunto de datos. Algunos lenguajes data-parallel restringen su uso sólo a arreglos, mientras otros extienden el concepto a conjuntos, listas y otras estructuras.

El modelo data-parallel se ajusta a arquitecturas con un único espacio de direcciones y memoria físicamente distribuida. De esta forma, la distribución de datos juega un rol primordial, pero las variables del programa son visibles a todos los procesos. Esto se refleja principalmente en las arquitecturas SIMD y NUMA, pero también existen implementaciones de alta performance para SMPs (multiprocesadores sincrónicos) y clusters [300].

La mayor ventaja del paralelismo de datos es la simplicidad de programa. Considerando la estructura global de un programa, hay sólo un flujo de control. El paralelismo se obtiene vía pasos data-parallel, los que son ejecutados uno después de otro, mezclados con pasos secuenciales.

El conjunto de datos se divide en subconjuntos (o *dominios*), cada uno de los cuales es asignado a un procesador diferente. La división (o distribución de datos) en la mayoría de los lenguajes debe ser especificada por el programador, quien puede elegir entre varias opciones específicas. La distribución no necesariamente se fija una vez y para siempre: la mayoría de los lenguajes permiten redistribución durante la ejecución.

En cada paso, la operación respectiva se aplica en paralelo a los distintos dominios de datos. Idealmente, los dominios son operados de manera independiente. En la realidad, varias operaciones involucran datos de distintos dominios, introduciendo comunicación. En una comunicación, los datos son replicados (esto es, el propietario mantiene los datos y envía una copia al procesador que lo requiere). Obviamente, no solo los datos sino las operaciones deben asignarse a los procesadores. Algunos compiladores adoptan la *regla owner computes*, en la cual una asignación es llevada a cabo por el procesador que es dueño de los datos. En algunos sistemas, el programador puede especificar explícitamente la asignación de la operación.

Algunos ejemplos de lenguajes data-parallel de estilo SIMD son C* (diseñado especialmente para la CM-2), MPL (MasPar Language) y Parallaxis (independiente de la arquitectura) [100]. Una desventaja de los lenguajes SIMD es su restricción a problemas con una estructura muy regular; debido a la sincronización implícita después de cada paso elemental, cualquier uso de paralelismo de tareas (si puede expresarse) lleva a ineficiencias.

Algunos de los lenguajes SIMD fueron eclipsados por el estilo SPMD o *loosely synchronous* (pues no existe sincronización luego de cada paso elemental). Generalmente la performance es mejor con SPMD, pero éste deja de lado el concepto de un único flujo de control, haciendo más dificultoso el diseño y verificación de programas debido a posibles *race conditions*. Los lenguajes HPF y HPF-2 son adecuados para el tratamiento en paralelo de arreglos.

HPF [205] da al programador control explícito sobre la distribución de los datos; aunque no está obligado a hacerlo siempre, ésto normalmente mejora la performance. El diseño de HPF fue guiado por el objetivo de liberar al programador de las tareas que puede realizar el compilador, pero dándole control sobre las tareas que no pueden ser manejadas automáticamente. El precio por combinar performance con facilidad de uso fue una restricción en la aplicabilidad: sólo es apropiado para aplicaciones con estructura regular, pues los constructores de distribución de datos se restringen a patrones regulares. Esto dio lugar al desarrollo en 1997 de HPF-2 [165]: mejora a HPF con características de lenguaje regular y un gran conjunto de extensiones. HPF-2 soporta la expresión de paralelismo de tareas de manera limitada: diferentes subconjuntos de procesadores pueden realizar distintas operaciones. Sin embargo, aún es insuficiente el soporte para aplicaciones irregulares.

El estilo de paralelismo de datos SPMD realiza operaciones complejas sobre los varios dominios de datos. Si los dominios están en sí mismos estructurados internamente, y si las operaciones son data-parallel internamente, se obtiene un nuevo modelo denominado *paralelismo de datos anidado*. Este estilo encuadra con el paradigma de *dividir y conquistar* descripto en la Seccion 3.5.3, y puede ayudar a codificar aplicaciones irregulares. Puede ser expresado en HPF y Fortran 90/95 [274]. Un lenguaje funcional anterior a éstos que adopta el paralelismo de datos anidado es NESL [42], desarrollado principalmente para propósitos de enseñanza y experimentación algorítmica.

### 3.2.3 Paralelismo de control

El *paralelismo de control* (también llamado paralelismo *de tareas*, *de grano grueso* o *funcional*) se refiere a la ejecución simultánea de distintos flujos de instrucciones. Estas instrucciones pueden ser aplicadas al mismo flujo de datos, pero típicamente se aplican a distintos. Se realiza una descomposición funcional y las tareas generadas ejecutan independientemente, comunicándose cuando lo necesitan.

Un ejemplo es el *pipelining*, donde la computación es paralelizada ejecutando un programa diferente en cada procesador y enviando los resultados intermedios al siguiente. Es un concepto sencillo, generalmente los bloques funcionales son fáciles de identificar, tiene bajo overhead y la estructura de comunicación es simple. Como contrapartida, los tiempos de procesamiento desiguales causan cargas dispares, el número de PEs es restringido (lo que bloquea la escalabilidad) y no hay balance de carga o es muy limitado.

El paralelismo funcional implica expresar el algoritmo como un grafo de dependencias generalizado, determinar la granularidad apropiada de operaciones y particionar las estructuras de datos a ser procesadas entre las replicaciones de las operaciones, elegir un lenguaje apropiado y representar el grafo de dependencias en ese lenguaje.

Los algoritmos para problemas que requieren paralelismo de control generalmente son mapeados adecuadamente en máquinas MIMD, dado que son necesarios múltiples flujos de instrucción. Las computadoras SIMD soportan sólo una corriente de instrucción y no pueden explotar eficientemente esta clase de paralelismo.

Mientras el modelo data-parallel ejecuta la misma operación sobre distintos datos, el paralelismo de tareas ejecuta distintas operaciones sobre los mismos o distintos datos. Normalmente es expresado en lenguajes no data-parallel, pero varios de estos permiten una forma limitada de paralelismo de tareas: poniendo las operaciones complejas dentro de un condicional, un programador puede forzar la ejecución paralela de distintas tareas en un paso data-parallel. Típicamente, el paralelismo de datos es de grano más fino que el de tareas. De esta forma, es más apropiado para arquitecturas fuertemente acopladas que para sistemas distribuidos con una red de interconexión lenta.

### 3.2.4 Paralelismo mixto

Muchos problemas exhiben una cierta cantidad tanto de paralelismo de datos como de control. La cantidad de paralelismo de tareas disponible en un problema usualmente es independiente del tamaño de éste y por lo tanto limitado; por el contrario, el paralelismo crece con el tamaño del problema. Por esto, para usar eficientemente un mayor número de procesadores es necesario explotar el paralelismo de datos inherente en la aplicación.

Debe notarse que no todas las aplicaciones paralelas de datos pueden implementarse usando lenguajes data-parallel ni todas pueden ser ejecutadas en máquinas SIMD; algunas son más apropiadas para MIMD. Esto se da en los casos en que los elementos de datos son generados dinámicamente de manera no estructurada, y la distribución de datos en procesadores debe realizarse también dinámicamente. Esta redistribución es posible en programas data-parallel solo a una escala global (esto es, no permitir que algunos procesadores sigan trabajando mientras otros se redistribuyen los datos).

Los paralelismos de datos y control pueden ser combinados para obtener una ex-

plotación simultánea de ambas características. En las aplicaciones de *paralelismo mixto*, varias computaciones data-parallel pueden ejecutarse concurrentemente de manera *task-parallel*. Este modelo emplea un estilo M-SIMD (*Multiple* SIMD) o M-SPMD (*Multiple* SPMD): es la combinación de paralelismo de tareas (MIMD o MPMD) y de datos (SIMD o SPMD). Su explotación tiene varias ventajas: una es la posibilidad de incrementar la escalabilidad dado que permite el uso de más paralelismo cuando se alcanza el máximo explotable de alguna de las dos clases. La integración de ambos modelos está lejos de ser fácil, principalmente a causa de tres problemas principales: diferencias en la estructura del programa paralelo, en la organización del espacio de direcciones y en la implementación del lenguaje. Estos temas son tratados en [28].

Gran parte de las investigaciones sobre la explotacion simultánea de paralelismo de datos y de tareas se realizaron en el área de los lenguajes de programación para dar accesos de alto nivel a más paralelismo en el área de compiladores, donde se estudian temas tales como scheduling y alocación de tareas data-parallel concurrentes [287]. En [281, 282], Ramaswamy introduce la estructura Macro Dataflow Graph (MDG) para describir programas de paralelismo mixto. En [61] se utiliza un modelo para estimar la mejora de performance debida a la utilización de paralelismo mixto para algunas aplicaciones científicas y se establecen cotas superiores para los beneficios obtenidos en grafos de tareas irregulares. En [90], el enfoque es usar un lenguaje secuencial como C con librerías de alta performance como ScaLAPACK [41] y una librería de comunicación asociada como BLACS; los procesadores disponibles son divididos explícitamente en subgrillas que reciben operaciones data-parallel a ejecutar.

En [109] se considera la generación de programas de paralelismo mixto y se discute cómo una separación clara en un nivel de tareas y uno de datos puede soportar el desarrollo de programas eficientes. El desarrollo comienza con una especificación del máximo grado de paralelismo de tareas y datos, y prosigue realizando varios pasos de derivación en el cual el grado de paralelismo es adaptado a una máquina paralela específica. Se muestra cómo son generados los programas y cómo puede establecerse la interacción entre ambos niveles. Se ilustra la utilidad del enfoque mediante ejemplos de análisis numérico que ofrecen el potencial de una ejecución mixta, pero para los cuales no es *a priori* claro cómo este potencial podría explotarse en una implementación sobre una máquina paralela específica.

Entre los lenguajes con alguna forma de soporte para paralelismo mixto se encuentran: NESL, UC (desarrollado por Caltech y UCLA e inspirado en lenguajes basados en conjuntos, permite que el programador especifique la sincronización de un conjunto de sentencias a niveles crecientes de granularidad [91, 26]), Fx (agrega directivas de paralelismo de tareas a un lenguaje data-parallel basado en Fortran 77 y HPF, dividiendo el conjunto de procesadores en subgrupos que pueden ejecutar computaciones data-parallel independientes [331]), Opus (agrega constructores de programación task-parallel a HPF para coordinar la ejecución concurrente de varios módulos data-parallel [64], permitiendo que

las tareas Opus sean de grano mucho más grueso comparado con otros lenguajes), Data-parallel Orca (extiende Orca con construcciones para paralelismo de datos, basándose en *objetos compartidos* [35]), Braid (extensión data-parallel del *Mentat Programming Language, MPL*, orientado a objetos basado en C++ [356]), etc.

### 3.2.5 Paralelismo de datos agregado

Uno de los desafíos es el desarrollo de lenguajes paralelos que minimicen el *tradeoff* entre performance y portabilidad, para permitir al programador escribir código que no asuma una arquitectura particular, al compilador optimizar el código de manera específica según la máquina, y al programador realizar ajuste ("*tuning*") de performance específico de la arquitectura sin extensas modificaciones al código fuente. En años recientes evolucionó un estilo de programación que puede denominarse *paralelismo de datos agregado*, caracterizado según [12] por:

- Paralelismo de datos. El paralelismo del programa proviene de ejecutar la misma función sobre varios elementos de una colección. Permite que el paralelismo crezca (o *escale*) con el número de elementos y procesadores. Las arquitecturas SIMD lo explotan a un grano muy fino.
- Ejecución agregada. El número de elementos de datos típicamente excede el número de procesadores, de modo que múltiples elementos son puestos en un procesador y manipulados secuencialmente. Esto trae beneficios como: reduce costos de comunicación, permite usar buenos algoritmos secuenciales localmente, los datos pueden pasarse en lotes entre los procesadores para amortizar el overhead de comunicación, y cuando la computación sobre un dato se demora esperando por comunicaciones pueden procesarse otros elementos.
- Sincronismo *loose*. Aunque el paralelismo de datos estricto aplica la misma función a cada elemento, las variaciones conceptuales en la naturaleza o posicionamiento de algunos elementos puede requerir distintas implementaciones de la misma función conceptual. Por ejemplo, los elementos del borde de un dominio computacional no tienen vecinos con los cuales comunicarse, pero el paralelismo de datos normalmente asume que los elementos interiores y exteriores deben tratarse igual. Estos casos excepcionales pueden manejarse fácilmente ejecutando una función distinta en los bordes.

Estas características hacen útil este estilo porque pueden obtenerse programas eficientes al ejecutarlos en máquinas MIMD. Pero, sin soporte de lenguajes, el estilo de programación promueve programas inflexibles embebiendo características críticas de performance como constantes (tales como el número e interconexión de procesadores, la

cantidad de elementos de datos, condiciones de borde, y sintaxis de comunicación específica del sistema). Si alguno de estos parámetros cambia, generalmente son necesarios numerosos cambios a estas cantidades fijas. Como consecuencia, se han introducido algunos lenguajes que soportan aspectos claves de este estilo. Sin embargo, a menos que todos los aspectos sean soportados, la performance, la escalabilidad, la portabilidad o el costo de desarrollo pueden verse seriamente afectados.

En [12] se brindan abstracciones que son implementables eficientemente sobre todas las arquitecturas MIMD, junto con mecanismos para manejar tipos comunes de paralelismo, distribución de datos y condiciones de borde. Se basa en un modelo de cómputo MIMD práctico llamado *Candidate Type Architecture* (CTA) [323]. Luego, se ocultan los aspectos de arquitectura que no son significativos, y se exponen y parametrizan de manera independiente de la arquitectura los que son esenciales para la performance. El resultado es el modelo de programación *Phase Abstractions*, que brinda control sobre la granularidad del paralelismo, el particionamiento de datos y un constructor paralelo híbrido de datos y función que soporta la descripción de las condiciones de borde.

## 3.3 Memoria compartida e intercambio de mensajes

### 3.3.1 Paralelismo con memoria compartida

En el modelo de memoria compartida, múltiples tareas (generalmente con distinta funcionalidad) ejecutan en paralelo comunicándose mediante escrituras y lecturas en una memoria común (que puede existir físicamente o estar implementada como abstracción para el programador). Típicamente, sólo algunas de las variables y estructuras de datos son compartidas entre las tareas y las otras son privadas (propiedad de una sola y no visibles al exterior). En algunos sistemas, las variables pueden compartirse entre un subconjunto de tareas.

La mayoría de las construcciones de programación de memoria compartida tratan con sincronización o manejo de tareas. No es necesario soporte para comunicación, sino que se expresa implícitamente (como por ejemplo en HPF). La programación tiene algunas características simples (la organización de memoria es la misma que en los modelos secuenciales, el programador no debe tratar con distribución de datos y no maneja detalles de comunicación como la alocación de buffers). Pero por otra parte debe tratar dificultades específicas como la sincronización (que afecta la performance) y la localidad (aunque no se requiere que el programador especifique la distribución de datos y el scheduling de tareas, debería tenerlos en cuenta por razones de eficiencia).

Este modelo se originó en la programación concurrente de uniprocesadores y luego se adaptó a los multiprocesadores sincrónicos (SMP), y es naturalmente apropiado para

éstos. Esta fue una gran motivación para el desarrollo de arquitecturas cc-NUMA, altamente escalables, que pueden correr software SMP existente aunque no con la misma performance. Se implementaron modelos de memoria compartida en SMPs, cc-NUMA, ncc-NUMA, máquinas paralelas de memoria distribuida y clusters, y pueden agregarse a las grillas en el futuro.

Especialmente en arquitecturas de memoria distribuida, la programación de memoria compartida compite con el modelo de pasaje de mensajes. Una gran diferencia entre ambos es el grado de transparencia. En memoria compartida, la ubicación de datos y tareas, y la replicación y migración son transparentes, mientras en el pasaje de mensajes no lo son. De esta forma, para el programador en el primer caso es más cómodo pero pierde control, los mecanismos del sistema funcionan bien en el caso general pero son subóptimos en casos especiales, y se dificulta la predicción de performance.

Dentro de este modelo pueden distinguirse variantes que difieren en las arquitecturas y aplicaciones para las que fueron diseñadas. Entre ellas se encuentran [228]:

- *Modelos de thread.* Son la clase más antigua y típica. Tratan con computaciones inherentemente paralelas y están bastante cercanas a los entornos secuenciales y SMPs. Los threads son más "livianos" y rápidos que los procesos. El modelo considera al programa como una colección de threads que cooperan para lograr un objetivo común, o comparten recursos de manera controlada por el programador. Su diseño estuvo influenciado por su uso frecuente en la programación de servidores. Entre los más importantes se encuentran Pthreads [229] y Java [265, 276].

- *Modelos de memoria compartida estructurada.* Son más adecuados para aplicaciones paralelas de grano fino que los modelos de thread. Un ejemplo es OpenMP [85], que opera en un nivel más alto que los threads (muchos compiladores OpenMP usan "C + Pthreads" como lenguaje de destino). Es un modelo de memoria compartida *estructurada* porque el lenguaje impone una estructura sobre el conjunto de tareas; mientras los modelos de thread permiten que cualquier thread cree o mate a casi cualquier otro, OpenMP fuerza una estructura jerárquica. Provee paralelismo de datos y de control, con énfasis en el primero y estilo de programación SMPD.

- *Modelos de memoria compartida distribuida.* Trata la abstracción de memoria compartida en arquitecturas con memoria físicamente distribuida (cc-NUMA, ncc-NUMA, clusters, grillas). Los sistemas *DSM* (*Distributed Shared Memory*) operan en bajo nivel, y pueden usarse tanto como interfases de programación de aplicaciones como lenguajes de destino para la compilación de sistemas de más alto nivel tales como OpenMP. Pueden ser implementados en hardware y/o software (por ejemplo, en el sistema operativo). Un mecanismo básico de sistema operativo es la *memoria virtual compartida*. Un tema crítico es el del mantenimiento de la consistencia. Uno de los sistemas DSM más conocidos es TreadMarks [15], implementado como una librería que corre en Unix y Windows NT.

- *Modelos de comunicación one-sided.* En este caso la comunicación es explícita. El programador debe usar distintos mecanismos para acceder las memorias locales y globales, y de esta forma conoce la diferencia de performance. La filosofía es distinta a los otros modelos, ya que no mantiene el objetivo de transparencia. Además, cae en la frontera entre la memoria compartida y el pasaje de mensajes. La memoria compartida (entre todos o algunos de los procesadores) debe ser alocada explícitamente por el programador, y todo el resto es memoria privada. Algunos facilitan el solapamiento entre cómputo y comunicación, y pueden ser implementados como librería o lenguaje. Ejemplos son MPI-2 y BSP (ambos principalmente sistemas de pasaje de mensajes).

El tema de memoria compartida es cubierto en numerosos libros de texto, tales como [181, 272].

Aunque el concepto de programación por memoria compartida se encuentre fuertemente asociado a la noción de threads, también es aplicable para procesos. Unix [327] brinda funciones mediante las cuales los procesos pueden acordar compartir una parte de sus espacios de direcciones, además de proveer mecanismos de sincronización y de creación y destrucción de procesos; permite obtener la misma funcionalidad que con Pthreads pero con mayor dificultad de codificación y peor performance. Ada 95 [57, 348] soporta comunicación por variables compartidas (coexistiendo con otros modelos); brinda un constructor tipo monitor para sincronización, similar al de Java.

Respecto de DSM, además de TreadMarks se han propuesto una cantidad de sistemas como CVM [200], Brazos [326], Milipede [186, 187], SciOS [204], Proteus [349], Sirocco [297], etc. Algunos aspectos interesantes incluyen soporte explícito para clusters de SMPs, integración con hardware SCI, migración de tareas y datos compartidos en unidades más pequeñas que páginas.

Linda [17] define un modelo de memoria compartida que impone una estructura de manera que la memoria es accedida en unidades específicas de la aplicación (tuplas de tamaño variable) en lugar de páginas de tamaño fijo. El modelo one-sided es utilizado en el lenguaje Split-C [83], que soporta programación data-parallel, de memoria compartida y pasaje de mensajes.

### 3.3.2 Paralelismo con pasaje de mensajes

Como se detalló en el Capítulo 1, en este modelo varios procesos ejecutan en paralelo y se comunican enviando y recibiendo mensajes. No tienen acceso a una memoria compartida, esto es, operan sobre espacios de direcciones disjuntos y toda la comunicación se reliza por intercambio explícito de mensajes. Las operaciones básicas son *send* y *receive*, las cuales presentan variantes.

El intercambio de mensajes puede servir a distintos propósitos. El más obvio es compartir datos entre un emisor y un receptor que se conocen y cuya interacción fue planeada por el programador. El segundo es el establecimiento de una conexión emisor/receptor no planeada de antemano (mediante un *receive* anónimo hecho por el receptor). Y el tercer propósito es la *sincronización*, que es un caso especial de comunicación. En las rutinas de emisión y recepción el programador debe especificar detalles de bajo nivel como la identidad del receptor, la dirección y tamaño de un buffer fuente o destino, y la longitud del mensaje.

El pasaje de mensajes soporta programación SPMD y MPMD. El dominio de aplicación típico es el cómputo científico orientado a performance. Aunque con frecuencia puede usarse para codificar aplicaciones data-parallel, hay diferencias fundamentales entre los modelos de datos paralelos y de mensajes: los primeros operan en un nivel mucho más alto (el programador no debe manejar detalles que son dejados al compilador) y el pasaje de mensajes adopta una visión localizada (el programador piensa en términos de los procesos individuales).

El modelo de pasaje de mensajes se basa claramente en la arquitectura de máquinas de memoria distribuida y por lo tanto es fácil de implementar; sin embargo opera a bajo nivel y tiene las siguientes consecuencias:

- *Programabilidad.* Impone una pesada carga al programador, que es responsable de manejar todos los detalles de distribución de datos y scheduling de tareas, así como la comunicación entre éstas.

- *Eficiencia.* Dado que todo está bajo el control del programador, éste puede lograr performance cercana al óptimo si le dedica suficiente tiempo al tuning. Puede redistribuir datos y procesos con cualquier patrón, replicar los datos en unidades de tamaño variable, etc.

- *Portabilidad.* Dado que el pasaje de mensajes fue establecido en forma temprana, se han desarrollado standards. Esto no garantiza portabilidad de performance.

Entre los principales sistemas de pasaje de mensajes se encuentran *Message Passing Interface* (MPI) [322] y *Parallel Virutual Machine* (PVM) [117]. Ambos siguen el enfoque de librería y de esta forma son sencillos para implementar. Las librerías están definidas en C, C++ y Fortran, además de existir extensiones a Java. Por otra parte, dentro de los lenguajes que han adoptado el modelo de mensajes se encuentra Occam [184], que implementa una gran parte de las características del CSP de Hoare.

### 3.3.3 Comparación entre ambos modelos

Un tema de debate es en qué casos cada uno de los modelos es más apropiado, y en este sentido, existen argumentos variados.

El modelo de memoria compartida tiene cierta ventaja en la *programabilidad* ya que, las estructuras de datos no necesitan ser divididas entre los procesos y tiene similitud con el estilo de programación *secuencial*, facilitando la transición a los programadores. Es más adecuado para la *paralelización incremental*; transformar un programa secuencial es uno de pasaje de mensajes es más abrupto. Sin embargo, en la programación de memoria compartida *orientada a la performance*, el programador debe tomar en cuenta la distribución de datos y computaciones, lo que debilita la primera de las ventajas. En el pasaje de mensajes, el programador puede establecer directamente sus intenciones mientras en memoria compartida también depende de transformaciones del compilador.

El pasaje de mensajes tiende a ser superior con respecto a performance, pero debe diferenciarse entre las distintas arquitecturas, aplicaciones y sistemas de programación. Por otra parte, la performance es más *predecible*, pues el texto del programa indica cuándo tiene lugar una comunicación y la distribución de datos y scheduling de tareas la hace el programador. Memoria compartida requiere atención más detallada a la sincronización y pasaje de mensajes a detalles de comunicación.

## 3.4 Esquema de clasificación de Skillicorn y Talia

Es un esquema reciente de clasificación de modelos de programación paralelos, propuesto en [319], que utiliza dos dimensiones. En la primera, los modelos son clasificados de acuerdo a su grado de abstracción, relacionado con el nivel en la jerarquía. Estrictamente hablando, el grado de abstracción refleja cuántas actividades de diseño de programa son responsabilidad del programador y cuántas de las herramientas. En esta dimensión se distinguen 6 clases:

- Paralelismo implícito. El programador no da ningún *hint* sobre la ejecución paralela del programa.
- Paralelismo explícito y descomposición implícita. El programador indica el potencial paralelismo, pero deja que las herramientas decidan si es explotado (es decir, que definan las tareas paralelas).
- Descomposición explícita pero mapeo implícito. El programador define las tareas, pero otras actividades se dejan a las herramientas.
- Mapeo explícito pero comunicación implícita. Como en la clase previa, pero el programador es responsable del scheduling de tareas.

- Comunicación explícita y sincronización implícita. El programador es responsable de la mayoría de las actividades; sólo la sincronización es soportada en algún grado.
- Diseño completo del programa manejado por el programador.

En la segunda dimensión, el esquema distingue entre tres clases:

- Número dinámico de procesos y sin límite en el volumen de comunicación (donde *dinámico* significa no conocido antes del tiempo de corrida).
- Número de procesos fijado estáticamente y sin límite en el volumen de comunicación (entendiendo por *estático* conocido en tiempo de compilación).
- Número de procesos fijado estáticamente y con límite del volumen de comunicación.

El esquema completo consta de un total de 18 clases, de las cuales 15 son ejemplificadas por modelos reales en [319].

## 3.5 Paradigmas de Computación Paralela

### 3.5.1 Maestro/Esclavo

Un paradigma algorítmico basado en organizaciones del mundo real es Maestro/Esclavo (*Master/Slave* o *Master/Worker*): un proceso (o procesador) actúa como maestro, dirigiendo a todos los otros que funcionan como esclavos. El master iterativamente envía datos a los workers y recibe resultados de éstos. Pueden distinguirse dos casos básicos de acuerdo a la dependencia entre las iteraciones:

- Iteraciones dependientes: el master necesita los resultados enviados por todos los workers para poder generar un nuevo conjunto de datos.
- Entradas de datos independientes: los datos arriban al maestro, y éste no necesita los resultados anteriores para poder generar nuevos conjuntos de datos.

Respecto de la distribución de los datos, existen dos opciones:

- Distribuir todos los datos disponibles: cuando el master tiene listo un conjunto de datos para enviárselo a los workers, los distribuye todos siguiendo alguna política determinada.

- Distribuir bajo petición: el master distribuye un subconjunto de datos a los workers, y a medida que éstos van terminando le piden más.

Por ejemplo, algunos enfoques de tipo *branch-and-bound* a problemas de optimización mantienen la mejor solución encontrada hasta el momento, así como una lista de los subproblemas que necesitan ser explorados. En una implementación master/slave, el maestro mantiene ambos ítems y es responsable de distribuir los subproblemas a los esclavos; éstos deben procesar los subproblemas y enviar los resultados al master (quien determina si es la mejor solución corriente), reportar nuevos subproblemas que necesitan ser explorados al maestro y notificarlo cuando están libres para trabajar en un nuevo subproblema.

Si bien existen muchas variantes, la idea básica es que un procesador es responsable de la coordinación general, y los otros de resolver los subproblemas asignados. Es una variación de SPMD donde hay dos programas en lugar de uno solo.

Un tema a considerar es el caso en que los procesadores donde son ejecutadas las tareas son heterogéneos y con distintas velocidades, lo que provoca dificultades en el balance de las cargas asignadas a cada elemento de procesamiento.

### 3.5.2 Pipeline y algoritmos sistólicos

El *pipelining* es una técnica algorítmica paralela basada en modelos que recuerdan una línea de montaje. Un problema grande, tal como analizar un número de imágenes, puede particionarse en una secuencia de pasos a realizar sobre cada una (por ejemplo, filtrado, etiquetado o *labeling* y análisis de escena). Si se cuenta con tres procesadores, y cada paso toma aproximadamente la misma cantidad de tiempo, se puede comenzar haciendo el filtrado sobre la primera imagen en el primer procesador. Luego esta primera imagen se pasa al siguiente procesador para etiquetar, mientras el primero comienza a filtrar la segunda. En el tercer paso, la imagen inicial está en el tercer procesador para análisis de escena, la segunda en el segundo procesador para labeling, y la tercera en el primer procesador para ser filtrada.

Este modelo se mapea naturalmente a una computadora paralela configurada como un arreglo lineal (esto es, *mesh* unidimensional o anillo sin conexión *wraparound*). Este escenario simple puede extenderse de varias formas. Por ejemplo, como en una línea de montaje real, los procesadores no necesitan ser todos idénticos y pueden estar optimizados para su tarea. Además, si una tarea tarda más que las otras, entonces pueden asignarse a ella más procesadores. Finalmente, el flujo puede no ser una línea simple. Por ejemplo, un proceso de ensamble de automóviles puede tener una línea trabajando en el chasis, otra en el motor, y eventualmente ambas son combinadas. Este pipelining generalizado es llamado *procesamiento sistólico*; por ejemplo, algunas operaciones matriciales y sobre imágenes pueden realizarse de manera sistólica bidimensional.

### 3.5.3 Dividir y conquistar

Este paradigma bien conocido en la programación secuencial puede usarse también para expresar paralelismo. Explota la división repetida de problemas y datos en subproblemas similares de menor tamaño (fase de *dividir*). Estos múltiples subproblemas son resueltos independientemente (fase de *conquistar*), con frecuencia de manera recursiva, y las soluciones son combinadas finalmente en la solución global. Por esto, estrictamente hablando, el método debería llamarse *dividir, conquistar y combinar*.

Esta metodología es útil en cómputo paralelo porque las subdivisiones lógicas en subproblemas pueden corresponder a descomposición física entre procesadores, donde eventualmente cada subproblema está contenido en un único procesador. A excepción del nodo raíz, todos los procesos restantes son iguales. Cada proceso recibe una fracción de datos; si puede procesarlos, los procesa; si no es así, crea un cierto número de "hijos" y les distribuye los datos.

Ejemplos clásicos de algoritmos que utilizan este enfoque son *Mergesort* y *Quicksort*. En el primero, la secuencia de entrada se divide en dos subsecuencias, que son ordenadas recursivamente y luego mezcladas. En este caso, el paso *dividir* es sencillo: las subsecuencias son simplemente la primera y segunda mitad de la entrada. El paso *combinar* es ligeramente más complicado pues requiere un procedimiento especial para *mezclar* las dos subsecuencias ordenadas. En Quicksort, la entrada es dividida en tres subsecuencias *independientes* consistentes de todos los elementos que son menores, iguales y mayores que un elemento *pivot*, respectivamente. La primera y tercera son ordenadas recursivamente y luego se concatenan las tres. A diferencia de Mergesort, el paso de *combinar* (concatenación) es trivial comparado con el de *dividir* (particionar la secuencia con respecto al pivot).

Puede argumentarse que *todos* los algoritmos paralelos se basan en alguna forma de dividir y conquistar ya que, por definición, de eso se trata el paralelismo: un problema computacional se divide en subproblemas que son resueltos simultáneamente usando varios procesadores. De algún modo, esto es verdad. Pero esta interpretación amplia es demasiado general para permitir una apreciación real de la potencia del método. No se trata sólo de una subdivisión directa de tareas entre procesadores: la eficiencia se deriva de la selección apropiada de los subproblemas y su asignación a procesadores específicos.

Este tema es tratado ampliamente en [7], incluyendo ejemplos de búsqueda de ítems en una lista, *merging*, selección del elemento menor de una secuencia, y cómputo del *convex hull* de un conjunto de puntos del plano. En [252] puede encontrarse una solución genérica al problema de *labeling* en imágenes utilizando un enfoque de dividir y conquistar. [52] contiene ejemplos de quicksort paralelo y cómputo de la transformada rápida de Fourier (FFT), además de un análisis de complejidad de las soluciones.

### 3.5.4 SPMD

Al tratar el paralelismo de datos se hizo mención al paradigma SPMD (*Single Program Multiple Data*), originalmente definido por Alan Karp en [196]. En este modelo el programador (o el supercompilador) genera un programa único que cada nodo ejecuta sobre una porción diferente del dominio de datos. La diferente evaluación de algún predicado en sentencias condicionales permite que cada nodo tome distintos caminos del programa. La implementación de un programa SPMD involucra dos grandes fases: elección de la distribución de datos y generación del programa paralelo [68].

La primera determina el lugar que ocuparán los datos en los nodos. Esto influye fuertemente en el balance de carga porque en este tipo de computaciones, el trabajo es directamente proporcional al número de datos asignado a cada nodo. Para computaciones regulares en máquinas uniformes, la elección de la mejor distribución no es complicada ya que se asumen nodos idénticos. Por otra parte, para computaciones irregulares, los requerimientos de balance necesitan distribuciones más complejas que sólo pueden decidirse en tiempo de corrida. Dificultades análogas ocurren cuando la plataforma es heterogénea.

La generación del programa convierte el algoritmo secuencial en el programa SPMD. En la mayoría de los lenguajes, esta fase depende de la distribución de datos elegida e involucra cuatro tareas principales: (1) Obtener información de nodo y entorno que es crucial para establecer el rol que cada nodo debe jugar en el programa; (2) Acceder los datos vía funciones *data-inquiry* que convierten índices *globales-a-locales* y *locales-a-globales*, extraer datos locales desde dominios globales, verificar si una entrada de datos es mantenida por un cierto nodo o subconjunto de nodos, etc.; (3) Insertar primitivas de pasaje de mensajes para intercambiar datos ubicados en el espacio de direcciones de otros nodos; (4) Realizar operaciones sobre datos directamente accesibles.

### 3.5.5 Operaciones globales o colectivas

Para manejar estructuras completas en un paso, es útil tener un conjunto de operaciones que realicen tales manipulaciones. Estas *operaciones globales* pueden ser muy dependientes del problema, pero algunas han resultado de suma utilidad. Por ejemplo, el *broadcast* es una operación global común, usada para enviar datos de un procesador a todos los otros. Algunas extensiones al broadcast incluyen realizarlo simultáneamente con distintos y predeterminados subconjuntos de procesadores (*broadcast basado en subconjuntos*). Las operaciones globales son soportadas en varios modelos como paralelismo de datos, memoria compartida, pasaje de mensajes y otros. Están cercanas en concepto al paralelismo de datos pues adoptan una visión global de la computación [23].

Algunas operaciones globales se definen en términos de un operador $\otimes$ (semigrupo asociativo y conmutativo). Algunos ejemplos incluyen *mínimo*, *máximo*, *or*, *and*, *suma*

y *producto*. Por ejemplo, si el objetivo es obtener el máximo de un conjunto de valores $V(i)$, el $\otimes$ representa al máximo y la operación de aplicar $\otimes$ a los $n$ valores se denomina *reducción*. Si se hace broadcast del valor de la reducción a todos los procesadores, esto se conoce con el nombre de *reporte*.

Las operaciones globales brindan una manera útil de describir las acciones mayores en programas paralelos. Además, muchas veces se encuentran disponibles en implementaciones altamente optimizadas. Las operaciones de reducción son tan importantes que muchos compiladores las detectan automáticamente aún cuando no tienen soporte explícito para otras.

Algunas importantes operaciones globales incluyen [23]:

- *Sort*. Dado un conjunto ordenado $X = \{x_0, \cdots, x_{n-1}\}$ tal que $x_i < x_{i+1}\ \forall\ 0 \leq i < n-1$ ($X$ es un subconjunto de un tipo de datos ordenado linealmente), y dado que los $n$ elementos de $X$ están distribuidos arbitrariamente en un conjunto de $p$ procesadores, la operación de *sort* (re)acomodará los miembros de $X$ de modo que estén ordenados con respecto a los procesadores. Esto es, luego de ordenar, $x_0, \cdots, x_{\lfloor n/p \rfloor}$ estarán en el primer procesador, $x_{\lfloor n/p \rfloor + 1}, \cdots, x_{2\lfloor n/p \rfloor}$ estarán en el segundo procesador, etc. Esto asume un ordenamiento de los procesadores, así como de los elementos.

- *Merge*. Si $D_1$ y $D_2$ son subconjuntos de algún tipo de datos ordenado linealmente, y cada uno está distribuido de manera ordenada entre conjuntos disjuntos de procesadores $\mathcal{P}_1$ y $\mathcal{P}_2$ respectivamente, entonces la operación *merge* combina $D_1$ y $D_2$ para obtener un único conjunto sorteado almacenado de manera ordenada en el conjunto de procesadores $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$.

- *Read/Write asociativo*. Estas operaciones comienzan con un conjunto de registros *master* indexados por claves únicas. En el *read* asociativo, cada procesador especifica una clave y termina con los datos en el registro maestro indexados por esa clave (si tal registro existe) o un flag indicando que no hay tal registro. En el *write* asociativo, cada procesador especifica una clave y un valor, y cada registro *master* es actualizado aplicando $\otimes$ a todos los valores enviados a él (los registros *master* son generados para todas las claves escritas). Estas operaciones son extensiones de las de CRCW *PRAM*: modelizan *PRAM* con memoria asociativa y una operación de combinación poderosa para escrituras concurrentes. En la mayoría de las máquinas con memoria distribuida, el tiempo para realizar estas operaciones está dentro de una constante multiplicativa del tiempo necesario para simular la lectura y escritura concurrente, y el uso de las operaciones más poderosas puede resultar en simplificaciones algorítmicas y mejoras de performance significativas.

- *Compresión*. Mueve datos a una región de la máquina donde es posible la comunicación interprocesador óptima. Por ejemplo, comprimir $k$ ítems dentro de una mesh bidimensional de grano fino los moverá a un subcuadrado de $\sqrt{k} \times \sqrt{k}$.

- *Prefijo Paralelo.* Dado un conjunto de valores $a_i$, $1 \leq i \leq n$, esta computación determina $p_i = a_1 \otimes a_2 \otimes \cdots \otimes a_i$, $\forall\ i$.

- *Broadcast All-to-all.* Dados los datos $D(i)$ en el procesador $i$, cada procesador recibe una copia de $D(i)$ $\forall\ i$.

- *Comunicación personalizada All-to-all.* Cada procesador $P_i$ tiene un ítem de datos $D(i, j)$ que es enviado al procesador $P_j$, $\forall\ i \neq j$.

Las computaciones de prefijo paralelo son de gran importancia en muchas aplicaciones. En algunas, el problema a ser resuelto puede reducirse a una computación de sumas prefijas; en otras, esta computación es un paso fundamental en la resolución del problema, por lo que son necesarios algoritmos eficientes. Las aplicaciones cubren diversas áreas, como estudio de secuencias, geometría computacional, teoría de grafos, cómputo combinatorio y análisis numérico. Puede encontrarse un análisis detallado en [7].

## 3.6 Diseño de algoritmos paralelos

El diseño de algoritmos paralelos no se reduce fácilmente a simples recetas, sino que requiere un pensamiento integrador con cierto grado de *creatividad* [112]. Sin embargo, puede beneficiarse por un enfoque metódico que maximice el rango de opciones consideradas, provea mecanismos para evaluar alternativas, y reduzca el costo de *backtracking* por malas elecciones.

La mayoría de los problemas tienen distintas soluciones paralelas, y la mejor puede diferir esencialmente de las secuenciales existentes. Una posibilidad de diseño [112] es considerar temas tales como independencia de la máquina y concurrencia en etapas iniciales, dejando aspectos específicos de la máquina para el final. Esta metodología estructura el proceso en cuatro etapas: *particionamiento* (descomposición de la computación en tareas), *comunicación* (estructura necesaria para coordinar la ejecución de tareas), *aglomeración* (evaluación de las tareas y estructura definidas con respecto a performance y costos de implementación, combinando tareas para mejorar) y *mapeo* (cada tarea es asignada a un procesador, estática o dinámicamente por algoritmos de balance de carga). En las dos primeras se enfoca la concurrencia y escalabilidad buscando algoritmos con estas cualidades; en la tercera y cuarta la atención se mueve a la localidad y temas relativos a la performance.

La salida del proceso de diseño puede ser un programa que crea y destruye tareas dinámicamente, usando técnicas de balance de carga para controlar el mapeo de tareas en procesadores. O puede ser un programa SPMD que crea exactamente una tarea por procesador. El mismo proceso puede aplicarse en ambos casos, aunque si el objetivo

es producir un programa SPMD, los temas de mapeo quedan incluidos en la fase de aglomeración. En las siguientes subsecciones se tratan en detalle cada una de las fases.

### 3.6.1 Particionamiento

Esta etapa intenta exponer oportunidades para la ejecución paralela. Por lo tanto, el foco está en definir un gran número de pequeñas tareas para obtener una descomposición *de grano fino*; esto brinda la mayor flexibilidad en términos de algoritmos paralelos potenciales.

Una buena partición divide en piezas pequeñas tanto la *computación* asociada a un problema como los *datos* sobre los cuales opera esta computación. Al diseñar una partición, los programadores generalmente primero enfocan los datos asociados al problema, luego determinan una división apropiada para éstos y finalmente asocian computación con datos (técnica de *descomposición de dominio*). El enfoque alternativo (*descomposición funcional*) primero descompone la computación a realizar y luego trata los datos. Son técnicas complementarias que pueden aplicarse a distintas componentes de un mismo problema o aún al mismo problema para obtener algoritmos diferentes.

En la primera etapa de diseño se busca evitar la replicación de computación y datos; este aspecto puede ser revisado luego, ya que puede valer la pena replicar alguno para reducir los requerimientos de comunicación.

#### Descomposición de dominio

Se busca descomponer primero los datos asociados con un problema, en lo posible en piezas de igual tamaño. Luego se particiona la computación, típicamente asociando cada operación con los datos sobre los que opera. Esto da un número de tareas, donde cada una comprende algunos datos y un conjunto de operaciones sobre ellos; una operación puede necesitar datos de varias tareas, y en este caso se requiere comunicación para mover los datos (este punto es tratado en el paso siguiente del proceso de diseño).

Los datos que son descompuestos pueden ser la entrada al programa, la salida computada, o valores intermedios. Son posibles distintas particiones de acuerdo a las estructuras de datos. Algunas reglas son enfocar primero la estructura de datos más grande, o la accedida más frecuentemente. Las fases de la computación pueden operar sobre distintas estructuras o demandar diferentes descomposiciones en cada fase. En este caso, se trata cada fase separadamente y luego se determina cómo son combinadas las descomposiciones y algoritmos paralelos desarrollados para cada una.

Existen diferentes métodos para descomposición de dominio, utilizables en aplicaciones regulares e irregulares. Algunos ejemplos son descomposición binaria, RCB (*recursive*

*coordinate bisection*), RGB (*recursive graph bisection*), RSB (*recursive spectral bisection*), ORB (*orthogonal recursive bisection*), HRB (*hierarchical recursive bisection*), distribución cíclica o *scattered*, algoritmos *greedy*, *simulated annealing*, etc. [37, 29, 161, 162, 198, 107, 227, 199], El tema ha merecido un extenso tratamiento en la literatura y es un área en constante evolución. Puede encontrarse abundante bibliografía en [87, 249, 346, 345, 190].

#### Descomposición funcional

Representa una manera diferente y complementaria de pensar los problemas. En este caso, el foco inicial está en la computación a ser realizada más que en los datos manipulados por la misma. Si se tiene éxito en dividir el cómputo en tareas disjuntas, se examinan los datos necesarios para ellas. Estos requerimientos pueden ser disjuntos (en cuyo caso la partición es completa) o pueden superponerse de manera significativa (haciendo necesaria considerable comunicación para evitar replicación de datos y dando una pista de posible conveniencia de descomponer el dominio).

El aporte de este enfoque está en que al apuntar a las computaciones puede descubrirse la estructura del problema, y posibilidades de optimización que no serían obvias a partir solamente del estudio de los datos. Por ejemplo, en una búsqueda de nodos solución en un árbol, el algoritmo no tiene ninguna estructura de datos que pueda descomponerse.

La descomposición funcional también juega un rol importante como técnica para estructurar el programa. Una descomposición que particiona no sólo la computación sino el código que la realiza puede reducir la complejidad de diseño global. Este es el caso de sistemas complejos que pueden estructurarse como conjuntos de modelos más simples conectados vía interfases. Algunas referencias útiles son [112, 214, 135].

### 3.6.2 Comunicación

Se intenta que las tareas generadas por una partición ejecuten concurrentemente, pero en general no son totalmente independientes. La computación a realizar en una tarea típicamente requiere datos asociados con otra que deben transferirse. Este flujo de información es especificado en la fase de *comunicación*. Esto puede realizarse en dos etapas: primero, definir una estructura de canales que enlacen directa o indirectamente las tareas que deben cooperar (productoras y consumidoras), y luego especificar los mensajes que viajarán sobre esos canales.

En problemas de descomposición de dominio estos requerimientos pueden ser dificultosos de determinar. En cambio, en los algoritmos paralelos obtenidos por descomposición funcional el proceso es casi directo: corresponden a flujos de datos entre tareas.

Existen distintas clasificaciones para los patrones de comunicación:

- Local/global. En la comunicación local, cada tarea se comunica con un conjunto pequeño de otras vecinas, mientras en la global lo hace con muchas.

- Estructurada/no estructurada. En la primera, una tarea y sus vecinos forman una estructura regular (como un árbol o una grilla), y en la segunda las redes de comunicación pueden ser grafos arbitrarios (por ejemplo, en métodos de elementos finitos la grilla computacional puede ser deformada para seguir un objeto irregular o para brindar mayor resolución en regiones críticas).

- Estática/dinámica. En la estática, la identidad de los patrones de comunicación no cambia en el tiempo; en la dinámica, las estructuras pueden ser variables y determinadas por datos computados en tiempo de corrida.

- Sincrónica/asincrónica. En la sincrónica, los productores y consumidores ejecutan de manera coordinada, con pares cooperando en operaciones de transferencia; la asincrónica puede requerir que un consumidor obtenga datos sin la cooperación del productor.

Los patrones de comunicación no estructurados generalmente no causan dificultades conceptuales en las primeras etapas del diseño, pero sí en la aglomeración y mapeo. En particular, pueden ser necesarios algoritmos sofisticados para determinar una estrategia de agrupamiento que cree tareas de aproximadamente igual tamaño y minimice la comunicación creando el menor número de aristas entre tareas. Si los requerimientos de comunicación son dinámicos, estos algoritmos deben ser aplicados con frecuencia durante la ejecución, lo que obliga a pesar su costo contra los beneficios.

### 3.6.3 Aglomeración

El algoritmo resultante de las etapas anteriores es aún abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina paralela particular. De hecho, puede ser altamente ineficiente si, por ejemplo, crea muchas más tareas que los procesadores disponibles y la máquina no está diseñada para ejecución eficiente de tareas chicas.

En la etapa de *aglomeración* se revisan las decisiones tomadas con la visión de obtener un algoritmo que ejecute de manera eficiente en una clase de máquina real. En particular, se considera si es útil combinar o *aglomerar* las tareas identificadas para obtener otras de mayor tamaño. También se determina si vale la pena *replicar* datos y/o computación. La cantidad de tareas al final de esta fase, aunque reducida, puede ser aún mayor que el número de procesadores; una opción es forzar la reducción a exactamente una tarea por procesador, por ejemplo si el procesador destino demanda un programa SPMD (con lo que también queda resuelto el problema del mapeo).

Hay tres objetivos, a veces conflictivos, que guían las decisiones de aglomeración y replicación: reducir los costos de *ingeniería de software*, retener la *flexibilidad* con respecto a las decisiones de escalabilidad y mapeo, y reducir los costos de comunicación incrementando la *granularidad* de cómputo y comunicación.

### Incremento de la granularidad

Un tema crítico que influye en la performance paralela es el costo de comunicación, que obliga a detener el cómputo para enviar y recibir mensajes. Una forma de mejorar la performance es reducir el tiempo utilizado para comunicarse; esto se logra de manera obvia enviando menos datos, pero también enviando menos mensajes (aunque los datos sean los mismos), ya que el costo de *startup* de mensaje influye fuertemente. De todas formas, también son importantes los costos de creación de tareas.

Si el número de *partners* de comunicación por tarea es chico, con frecuencia se puede reducir tanto el número de operaciones como el volumen de comunicación incrementando la granularidad de la partición, agrupando varias tareas en una. Esto es conocido como *efecto superficie-a-volumen*: los requerimientos de comunicación de una tarea son proporcionales a la superficie del subdominio sobre el que opera, mientras los de computación son proporcionales al volumen del subdominio. En un problema bidimensional, la "superficie" escala con el tamaño del problema mientras el "volumen" lo hace con el tamaño del problema agrupado. Por lo tanto, la cantidad de comunicación realizada para una unidad de computación (*cociente comunicación/computación*) decrece cuando el tamaño de tarea crece.

### Preservación de la flexibilidad

Al aglomerar tareas es fácil tomar decisiones que limiten innecesariamente la escalabilidad de un algoritmo. Por ejemplo, se podría elegir descomponer una estructura de datos multidimensional en una sola dimensión, pensando en que brinda la concurrencia suficiente para el número de procesadores disponible. Pero esto no tomaría en cuenta el hecho de que podría portarse a una máquina paralela más grande, limitando la eficiencia.

La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable. Si las tareas se bloquean con frecuencia esperando datos, puede ser ventajoso mapear varias tareas en un procesador y hacer *overlapping* de cómputo y comunicación. Otro beneficio de crear más tareas que procesadores es que esto brinda mayor alcance para las estrategias de mapeo que balancean la carga computacional sobre los procesadores disponibles.

El número óptimo de tareas está determinado típicamente por una combinación de modelización analítica y estudios empíricos. La importancia está en que el diseño no

incorpore límites innecesarios sobre el número de tareas a ser creadas.

#### Reducción de costos de ingenería de software

Un ítem importante, especialmente al paralelizar códigos secuenciales existentes, son los costos de desarrollo relativos asociados con las distintas estrategias de particionamiento. Desde esta perspectiva, las más interesantes pueden ser las que evitan cambios extensivos (por ejemplo, permitiendo la reutilización de rutinas existentes).

Con frecuencia, el algoritmo paralelo que se está diseñando debe ejecutar como parte de un sistema más grande. En este caso, otro punto a tener en cuenta son las distribuciones de datos utilizadas por otras componentes de programa.

### 3.6.4 Mapeo

En esta última etapa se especifica dónde ejecuta cada tarea. Este problema no existe en uniprocesadores o computadoras de memoria compartida que brindan *scheduling* de tareas automático. Generalmente el objetivo del mapeo es minimizar el tiempo de ejecución total, y pueden usarse dos estrategias: ubicar tareas que son capaces de ejecutar concurrentemente sobre *diferentes* procesadores para mejorar la concurrencia, o poner las tareas que se comunican frecuentemente en el *mismo* procesador para incrementar la localidad.

Claramente estas estrategias con frecuencia conflictúan, por lo que el diseño incluirá *tradeoffs*. Además, las limitaciones de recursos pueden restringir el número de tareas en un procesador. El problema de mapeo es *NP-completo*, esto es, no existe un algoritmo (de tiempo polinomial) tratable computacionalmente para evaluar estos tradeoffs en el caso general; sin embargo, existe considerable conocimiento en estrategias, heurísticas y las clases de problema donde son efectivas.

Muchos algoritmos desarrollados usando técnicas de descomposición de dominio establecen un número fijo de tareas de igual tamaño y comunicación local y global estructurada. En tales casos, un mapeo eficiente es directo. Se mapean tareas de modo que se minimice la comunicación interprocesador; también se puede elegir aglomerar tareas mapeadas al mismo procesador, si esto ya no se hizo, para obtener una tarea de grano grueso por procesador.

En algoritmos con descomposición de dominio más complejos, con cantidades de trabajo variables por tarea y/o patrones de comunicación no estructurados, las estrategias de aglomeración y mapeo pueden no ser tan obvias. Por lo tanto, se utilizan algoritmos de *balance de carga*, que muchas veces emplean técnicas heurísticas.

Los problemas más complejos son aquellos en que el número de tareas o la cantidad de computación o comunicación por tarea cambia dinámicamente durante la ejecución del programa. En el caso de los problemas desarrollados usando técnicas de descomposición de dominio, se puede usar una estrategia de *balance de carga dinámico* en la cual un algoritmo de balanceo es ejecutado periódicamente para determinar una nueva aglomeración y mapeo.

Los algoritmos basados en descomposición funcional con frecuencia contienen computaciones consistentes en muchas tareas de corta vida que coordinan con otras al comienzo y fin de la ejecución. En este caso, pueden usarse algoritmos de *scheduling de tareas*, que aloca tareas a procesadores que están ociosos o poco ocupados.

## 3.7 Areas de aplicación

Las áreas de aplicación del cómputo paralelo son muchas y muy variadas. No está dentro del alcance de esta Tesis un análisis exhaustivo de dichos dominios, pero a modo de ejemplo pueden citarse algunos tipos de problemas que pueden enfocarse con el paradigma paralelo:

- Tratamiento de imágenes digitales, con aplicaciones médicas (tomografías computadas), militares, visión por computadora, etc.
- Simulación de fenómenos físicos y químicos. Dinámica molecular, física de particulas, análisis de estructuras de proteínas.
- Métodos de análisis de elementos finitos (simulación de formaciones de metal en ciencias de materiales e ingeniería).
- Procesamiento de datos recogidos por satélites, sensado remoto.
- Oceanografía (por ejemplo, simulación de circulación oceánica).
- Aerodinámica computacional (simulación de túneles de viento, diseño y dinámica de vehículos).
- Problemas N-body (simulación del movimiento de $N$ partículas bajo la influencia de campos de fuerza mutua, basado en una ley de cuadrado inverso).
- Reconocimiento de patrones en secuencias de ADN.
- Problemas de optimización discreta.
- Inteligencia artificial, aprendizaje en redes neuronales.

- Procesamiento de consultas en grandes bases de datos.
- Predicción del clima, monitoreo de contaminación.
- Procesamiento de lenguaje natural, reconocimiento de voz.

La mayoría de estos son considerados problemas *Grand Challenge* [144]: problemas fundamentales en ciencia o ingeniería con un gran impacto económico y científico, y cuya solución puede obtenerse aplicando técnicas y recursos de computación de alta performance. Resolverlos implica performance en el rango del teraflop (un trillón de operaciones de punto flotante por segundo) y grandes volúmenes de memoria (del orden de 100 Gygabytes).

Las aplicaciones que se desprenden de estas áreas llevan, en muchos casos, a algoritmos comunes. Entre otros, pueden nombrarse [226, 209, 7]:

- Algoritmos de manejo de secuencias y arreglos (ordenación, *merging*, búsqueda, reducciones, etc.)
- Algoritmos matriciales (multiplicación, transposición, resolución de sistemas de ecuaciones lineales, *eigenvalues*, etc.)
- Problemas en árboles y grafos (búsquedas, camino mínimo, *spanning tree*, clausura transitiva, componentes conectadas, ruteo de paquetes, etc.)
- Algoritmos de búsqueda para problemas de optimización discreta (*depth first search* paralelo, *best first search* paralelo)
- Programación dinámica (formulaciones monádicas y poliádicas)
- Transformada rápida de Fourier
- Problemas de *scheduling*

Para cada una de estas clases de algoritmos existen numerosas aproximaciones, que en muchos casos dependen fuertemente de la arquitectura de soporte. Es importante el análisis de las soluciones integralmente, apuntando al *sistema paralelo* como combinación de programa y máquina de destino.

# Capítulo 4

# Métricas del Paralelismo

## 4.1 Introducción

En el mundo serial, la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa. Para propósitos prácticos, los requerimientos de memoria son importantes solo para que haya suficiente espacio disponible para resolver instancias del tamaño deseado; en la mayoría de las casos no hay beneficios por usar menos memoria, más allá de los económicos. Esta suposición mantiene como métrica de performance sólo al tiempo.

Cuando se utiliza un algoritmo paralelo en la resolución de un problema, interesa saber cuál es la ganancia en performance obtenida. En la computación paralela, como en la serial, las métricas dominantes son tiempo y memoria. Entre métodos alternativos que usan diferentes cantidades de memoria, preferiríamos el más rápido. Esto es, no hay ventaja por usar menos memoria a menos que ese menor uso resulte en una reducción del tiempo. Así, el tiempo de ejecución o complejidad en tiempo de un programa paralelo sigue siendo una métrica importante [209, 7, 182, 292], aunque existen otras medidas que pueden tenerse en cuenta en el mundo paralelo siempre que favorezcan a sistemas con mejor tiempo de ejecución.

A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución de un algoritmo depende no sólo del tamaño de su entrada sino también de la arquitectura de la computadora paralela y el número de procesadores. Por lo tanto, un algoritmo paralelo no puede ser evaluado aisladamente de la máquina de destino. Un *sistema paralelo* es la combinación de un algoritmo y la arquitectura paralela sobre la cual está implementado.

La diversidad torna complicado el análisis de performance del sistema paralelo. Uno debe preguntarse, por ejemplo *¿qué interesa medir?*, *¿qué indica que un sistema paralelo es mejor que otro?*, *¿qué sucede si se utiliza un algoritmo con mayor cantidad de proce-*

*sadores?*, etc.

En la medición de performance paralela es usual elegir un problema y testear el tiempo de ejecución variando la cantidad de procesadores. En este modelo subyacen definiciones de *speedup* y *eficiencia*, y argumentos contra el procesamiento paralelo tales como la *ley de Amdahl*. Los modelos de tiempo fijo usan el tamaño de problema como la figura de mérito. Los análisis y experimentos basados en tiempo fijo en lugar de tamaño fijo trajeron consecuencias sorprendentes mostradas en [149].

En el estado actual de la tecnología, es posible construir computadoras paralelas que emplean miles de procesadores, y la disponibilidad de tales sistemas llevó a interesarse en la performance los mismos. Al resolver un problema en paralelo es razonable esperar una reducción en el tiempo de ejecución que sea proporcional a la cantidad de recursos de procesamiento empleados. La *escalabilidad* de un algoritmo paralelo sobre una arquitectura paralela es una medida de su capacidad de usar efectivamente un número creciente de procesadores. El análisis de escalabilidad de una combinación algoritmo-arquitectura paralela puede usarse para una variedad de propósitos expresados en [211], como caracterizar la cantidad de paralelismo inherente en un algoritmo paralelo, o estudiar el comportamiento con respecto a cambios en parámetros de hardware tales como la velocidad de los procesadores y canales de comunicación.

En este Capítulo se presentan las métricas más utilizadas en la evaluación de sistemas paralelos. De todas formas, es interesante analizar la performance de diferentes clases de aplicaciones en distintas arquitecturas reales con el fin de testear el ajuste de las predicciones a la realidad. Esto se debe a que muchos sistemas paralelos no alcanzan su capacidad teórica, y las causas de esta degradación son muchas y variadas. En parte puede deberse a la poca correspondencia entre aplicaciones, software y hardware, pero existen otros factores que pueden influir y no siempre son tenidos en cuenta al formular una métrica. Entre ellos están: desbalance en la carga de trabajo de procesos y procesadores, demoras debidas a la sincronización, overhead por scheduling, topología, efecto de la granularidad, grado de escalabilidad del sistema, mapeo de procesos y datos a procesadores que puede acarrear mayor o menor comunicación, distintos niveles de memoria involucrados, etc. El análisis sobre plataformas disponibles permite estudiar el impacto que tienen algunos de estos factores sobre las implementaciones, y adecuar las métricas a las mismas.

## 4.2 Medidas de performance standard

Históricamente se han usado medidas tales como *MIPS* (millones de instrucciones por segundo) o *MFlops* (millones de instrucciones de punto flotante por segundo) para describir respectivamente la velocidad de ejecución de instrucción y la capacidad de punto flotante de una computadora paralela. En la práctica, estos y otros indicadores deberían ser

medidos corriendo benchmarks o programas reales en máquinas reales [182].

La mayoría de los fabricantes fijan la performance pico o sostenida en términos de MIPS o MFlops. Estas velocidades no son concluyentes, pues la performance real es siempre dependiente del programa o manejada por la aplicación. En general, la velocidad en MIPS depende del conjunto de instrucciones, varía entre programas, y varía inversamente con respecto a la performance, como observaron Hennesy y Patterson en 1990.

Comparar procesadores con distintos ciclos de reloj y conjuntos de instrucciones no es totalmente justo. Además del MIPS nativo, se puede definir un MIPS relativo con respecto a una máquina de referencia. De manera similar, los MFlops dependen del diseño de hardware y del comportamiento del programa. El MFlops nativo no distingue operaciones de punto flotante normalizadas y no normalizadas. Por ejemplo, una operación de división de punto flotante real puede corresponder a 4 operaciones de división de punto flotante normalizado. Se necesita usar una tabla de conversión entre operaciones reales y normalizadas para convertir un *rating* MFlops nativo a uno normalizado.

Existen otros indicadores tales como *Dhrystone* (mezcla unas 100 instrucciones de lenguajes de alto nivel y tipos de datos de aplicaciones donde no se usan operaciones de punto flotante), *Whestone* (Fortran-based, apunta a la performance de punto flotante; incluye operaciones enteras y de punto flotante que involucran indexación de arreglos, llamados a subrutina, pasaje de parámetros, *branching* condicional y funciones trigonométricas), *TPS* (transacciones por segundo; cada transacción puede involucrar una búsqueda en base de datos, respuesta a consulta, y operaciones de actualización), *KLIPS* (kilo-inferencias lógicas por segundo, utilizado para indicar la potencia de razonamiento de una máquina de Inteligencia Artificial), etc.

Una tendencia es no basar la evaluación de performance paralela en estos indicadores, ya que se podría contar con un gran número de MFlops y aún así tener baja performance en las aplicaciones [143]. El tema de la evaluación de performance de computadoras paralelas es tratado en numerosos artículos. Si bien este punto se encuentra fuera del alcance de esta Tesis, sólo a modo de ejemplo pueden citarse [143, 96, 151, 148, 154, 152, 153, 335, 150, 155, 156, 258, 270, 309, 338, 320, 126, 362]

## 4.3 Algunas definiciones y conceptos básicos

### 4.3.1 Sistema paralelo

Es la combinación de una arquitectura paralela y un algoritmo paralelo implementado en ella. En la mayoría de los casos se asume que la máquina paralela es un ensamble homogéneo, esto es, todos los procesadores y canales de comunicación son idénticos en velocidad.

### 4.3.2 Tamaño del problema

Una manera de expresar el tamaño del problema es representarlo por un parámetro del tamaño de la entrada. Por ejemplo, para un problema con matrices de $n \times n$ el tamaño del problema puede denotarse por $n$. Una desventaja de esta definición es que la interpretación de tamaño de problema cambia de un problema a otro. Por ejemplo, duplicar el tamaño de la entrada resulta en un incremento de 8 veces en el tiempo de ejecución serial para la multiplicación de matrices y de 4 veces para la suma.

Una definición más correcta es tal que independientemente del problema, duplicar el tamaño siempre signifique realizar dos veces la cantidad de computación. Por lo tanto, se define el *tamaño del problema* ($W$) como una medida del número total de operaciones básicas necesarias para resolverlo. Dado que puede haber varios algoritmos distintos para resolver el mismo problema, para mantener único el tamaño se lo define como el número de operaciones básicas requeridas por el algoritmo secuencial conocido más rápido en un solo procesador (complejidad de tiempo secuencial) [211]. Dado que se define en términos de la complejidad de tiempo secuencial, el tamaño del problema es una función del tamaño de la entrada.

Si el tiempo tomado por un algoritmo secuencial óptimo (o el más rápido conocido) para resolver un problema de tamaño $W$ en un solo procesador es $T_s$, entonces $T_s \propto W$, o $T_s = t_c W$, donde $t_c$ es una constante dependiente de la máquina que representa el costo de ejecutar cada operación.

### 4.3.3 Fracción serial

Se denomina *fracción serial* $s$ al cociente entre el tiempo de la componente serial de un algoritmo y su tiempo de ejecución sobre un procesador. La componente serial es aquella parte que no puede ser paralelizada y debe ejecutarse en un solo procesador.

### 4.3.4 Tiempo de ejecución paralelo

El *tiempo de ejecución paralelo* ($T_p$) es el tiempo transcurrido desde el momento en que comienza una computación paralela hasta que el último procesador termina su ejecución. Para un sistema paralelo dado, $T_p$ normalmente es una función del tamaño del problema ($W$) y el número de procesadores ($p$), y suele escribirse como $T_p(W, p)$.

## 4.4 Speedup

Una de las mediciones de performance más usadas en el dominio paralelo intenta describir cuánto más rápido corre la aplicación sobre una máquina paralela. En otras palabras, cuál es el beneficio derivado del uso de paralelismo, o cuál es el *speedup* que resulta.

El speedup ($S$) es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido ($T_s$) y el tiempo de ejecución paralelo del algoritmo elegido ($T_p$):

$$S = \frac{T_s}{T_p} \tag{4.1}$$

Como se expresa en [292], existe una diversidad en las definiciones de tiempos de ejecución serial y paralelo, que resultan en, al menos, 5 definiciones distintas de speedup.

En el *speedup relativo* [336, 335, 337], el tiempo serial utilizado es el tiempo de ejecución del programa paralelo cuando corre en un solo procesador de la máquina paralela. Luego, el speedup relativo obtenido por el programa paralelo $Q$ para resolver una instancia $I$ de tamaño $n$ usando $p$ procesadores es:

$$Speedup\,Relativo(I,p) = \frac{tiempo\,para\,resolver\,I\,usando\,Q\,y\,1\,procesador}{tiempo\,para\,resolver\,I\,usando\,Q\,y\,p\,procesadores}$$

El speedup relativo de un sistema paralelo no es un número fijo: depende de las características (no sólo el tamaño) de la instancia $I$ que se está resolviendo y del tamaño del computador paralelo. Por ejemplo, el tiempo para ordenar $n$ números usando cualquier método conocido depende no sólo de $n$ sino también del orden inicial de los números. Luego, se puede calificar al speedup relativo como *máximo*, *promedio* o *esperado*, y *mínimo*. Para cualquier combinación de tamaños de instancia y computador paralelo, el speedup relativo máximo está definido como

$$Speedup\,Relativo\,Máximo(n,p) = \max_{|I|=n}\{Speedup\,Relativo(I,p)\}$$

donde $|I|$ es el tamaño de la instancia $I$. De manera similar se definen el speedup relativo promedio y mínimo.

Para aquellos casos en que el tiempo de ejecución está determinado unívocamente por el número de entradas en una instancia no se hace distinción entre los diferentes speedups relativos. En ese caso puede utilizarse la notación $Speedup\,Relativo(n,p)$ para denotar el speedup relativo obtenido sobre instancias de tamaño $n$ cuando se usan $p$ procesadores.

En [188, 209, 284], el tiempo de ejecución paralelo es comparado con el tiempo que necesita el algoritmo/programa serial más rápido para la aplicación sobre un único procesador del computador paralelo. Dado que para muchas aplicaciones puede no conocerse el algoritmo más rápido, y que para otras puede no existir un algoritmo más rápido *para todas* las instancias, se toma como referencia el tiempo de ejecución del algoritmo secuencial usado en la práctica en lugar del *runtime* del algoritmo más rápido. El speedup resultante se denomina *speedup real*.

$$Speedup\ Real(I,p) = \frac{tiempo\ p/\ resolver\ I\ con\ el\ mejor\ prog.\ serial\ y\ 1\ procesador}{tiempo\ para\ resolver\ I\ usando\ Q\ y\ p\ procesadores}$$

Se puede definir $Speedup\ Real\ Máximo(n,p)$, $Speedup\ Real(n,p)$, etc., de la misma manera que para el relativo.

En el *speedup absoluto* [336, 335, 337], el tiempo de ejecución paralela se compara con el del algoritmo secuencial más rápido corriendo sobre la computadora serial más rápida. Como en el caso del speedup real, en realidad, la comparación se hace con respecto al algoritmo serial usado en la práctica.

$$Sp\ Absoluto(I,p) = \frac{tiempo\ p/\ resolver\ I\ con\ el\ mejor\ prog.\ serial\ y\ el\ proc.\ más\ rápido}{tiempo\ para\ resolver\ I\ usando\ Q\ y\ p\ procesadores}$$

Si $t_{mejor\ serial}(n)$ es la complejidad asintótica del mejor algoritmo serial para el problema, y $t^Q_{paralelo}(n)$ es la complejidad asintótica del algoritmo paralelo $Q$ bajo la suposición de que la máquina paralela tiene disponibles tantos procesadores como necesita, entonces el *speedup real asintótico* [226] se define como

$$Speedup\ Real\ Asintótico(n) = \frac{t_{mejor\ serial}(n)}{t^Q_{paralelo}(n)}$$

En problemas tales como *sorting*, donde la complejidad asintótica no está unívocamente caracterizada por el tamaño de instancia $n$, se usa la complejidad del peor caso.

El *speedup relativo asintótico* [226] difiere del asintótico real en que para la complejidad serial se utiliza la complejidad en tiempo asintótica del algoritmo paralelo corriendo en un solo procesador. A diferencia de las otras 3 medidas de speedup, el asintótico no es función del número de procesadores disponibles en el sistema paralelo, ya que se asume una cantidad ilimitada de ellos.

Para computar el speedup *analítico* (relativo, real o absoluto) del sistema paralelo, se pueden usar complejidades de tiempo analíticas como en [88, 212, 211, 226, 363, 364]. Para obtener el speedup *medido* se utilizan tiempos de ejecución reales, como en [336, 335, 363, 364]. En [292] pueden encontrarse ejemplos de cálculo de cada uno de los speedups descriptos.

### 4.4.1 Rango de valores de speedup

Si para resolver el problema utilizando una máquina paralela se utilizan $p$ procesadores, entonces el rango de valores de speedup *en general* está entre 0 y $p$.

Teóricamente, el speedup no puede exceder el número de procesadores. Si el mejor algoritmo secuencial toma $T_s$ unidades de tiempo para resolver un problema dado sobre un procesador, entonces puede obtenerse un speedup de $p$ en $p$ procesadores si ninguno de ellos gasta más de $T_s/p$ tiempo. Un speedup mayor que $p$ es posible solo si cada procesador gasta menos de $T_s/p$ tiempo para resolver el problema. En ese caso, un solo procesador podría emular los $p$ procesadores y resolver el problema en menos de $T_s$ unidades de tiempo. Esto es una contradicción porque el speedup, por definición, se computa con respecto al mejor algoritmo secuencial. Si $T_s$ es el tiempo de ejecución de ese algoritmo serial, entonces el problema no puede ser resuelto en menos de tiempo $T_s$ en un solo procesador. Para los casos donde el speedup puede superar a $p$, ver la Sección 4.4.3

La gráfica de esta métrica, denominada *curva de speedup*, refleja en las abcisas el número de procesadores y en las ordenadas el speedup obtenido.

Cuando al utilizar $p$ procesadores se logra una ganancia de $p$ se dice que se tiene un caso de *speedup perfecto* o *speedup lineal*. Esto se da en casos especiales, ya que normalmente la paralelización introduce factores de overhead (como comunicaciones, distribución de datos, etc.) que degradan el speedup.

El *máximo número de procesadores usables* ($p_{max}$) es el número de procesadores que da el máximo speedup $S^{max}$ para un $W$ dado. En ese caso, el usar más procesadores no incrementa el speedup.

**Ejemplo 4.1** *Suma de n números en un hipercubo n-procesador.*

Inicialmente se asigna un número a cada procesador, y al final uno de los procesadores almacena la suma. Asumiendo $n$ potencia de 2, esto puede hacerse en un hipercubo o un multiprocesador de memoria compartida en $\log n$ pasos, donde cada paso consta de una suma y la comunicación de una palabra. Los procesadores que se comunican están directamente conectados en un hipercubo (sus labels difieren en un bit). Tanto la suma como la comunicación toman una cantidad constante de tiempo, por lo que $T_p = \Theta(\log n)$. Dado que el problema puede ser resuelto en $\Theta(n)$ en un procesador, su speedup es $S = \Theta(n/\log n)$. La Figura 4.1 muestra el cómputo para 16 números en 16 procesadores [209]. □

Con frecuencia, además de saber cuánto más rápido corre el sistema paralelo se quiere tener una medida del costo al cual se obtiene esta mejora. En [292] se hace referencia a la métrica de *speedup de costo normalizado* y se la define como:

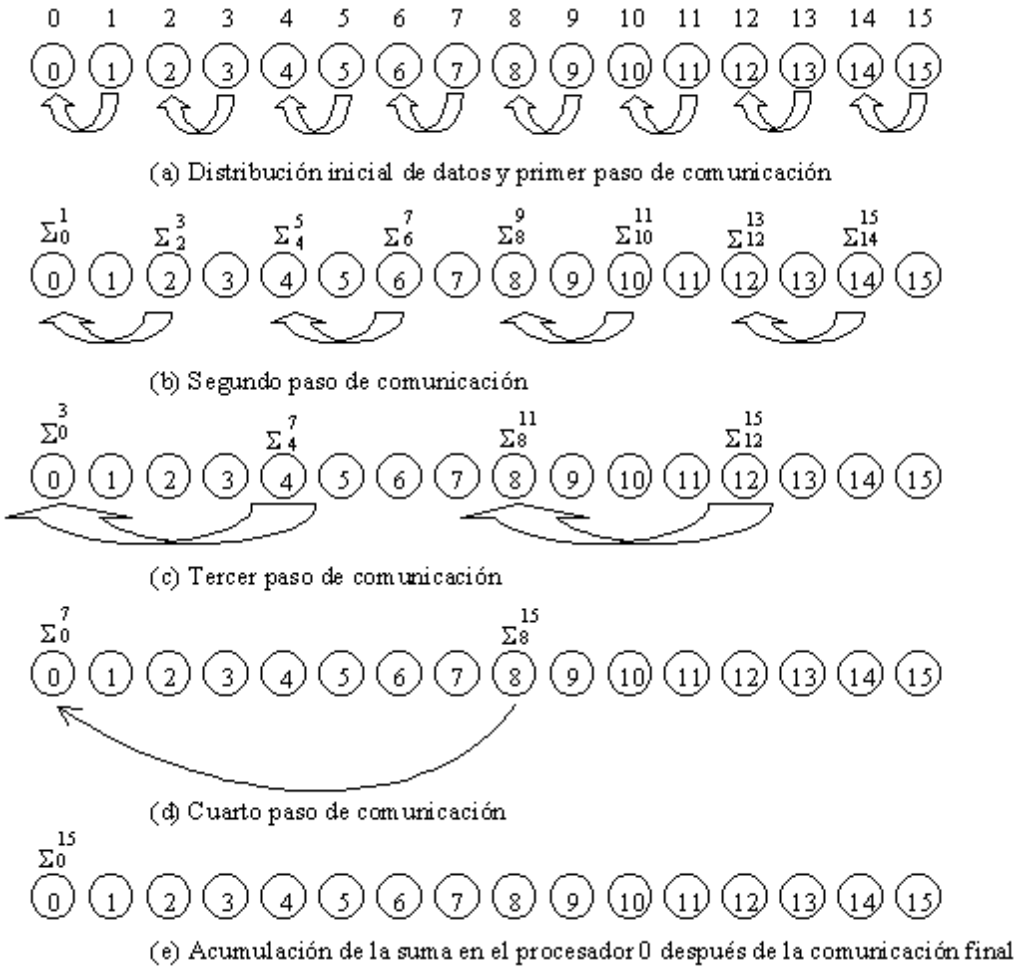


Figura 4.1: Suma de 16 números en un hipercubo 16-procesador

$$Speedup\ de\ costo\ normalizado(I,p) = \frac{speedup(I,p)}{(costo\ sistema\ paralelo) * (costo\ sistema\ serial)}$$

Si bien es un indicador atractivo, las dificultades encontradas para determinar cada uno de los costos hicieron que no se haya utilizado en forma extensa. En [365] se presenta el *costup* como una métrica que utiliza una combinación del speedup para indicar la efectividad en costo del sistema paralelo, aunque no incluye completamente los costos de software.

### 4.4.2 Factores que limitan el speedup

Existen una serie de factores que limitan el speedup, entre los que se encuentran:

- *Entrada/salida.* El porcentaje de entrada/salida es alto comparado con la cantidad de computación.
- *Algoritmo.* El algoritmo numérico no es adecuado para una máquina paralela, y es necesario reemplazarlo con un algoritmo paralelo.
- *Excesiva contención de memoria.* Es necesario rediseñar el código tendiendo a la localidad de datos. También pueden ayudar las técnicas de reutilización de cache.
- *Tamaño del problema.* El problema es demasiado chico para tomar ventaja de la ejecución paralela, o es fijo y no crece si se usan más procesadores.
- *Desbalance de carga.* Los procesadores tienen cargas de trabajo desiguales, lo que causa que algunos estén ociosos mientras esperan que otros terminen su trabajo.
- *Alto porcentaje de código secuencial.* Esto lleva a la *ley de Amdahl.*
- *Overhead paralelo.* Ciclos adicionales de CPU para crear regiones paralelas, crear threads, sincronizar, etc.

### 4.4.3 Speedup superlineal

En la práctica, a veces puede observarse un speedup mayor que $p$; este fenómeno es llamado *speedup superlineal.* Se debe usualmente a un algoritmo secuencial no óptimo o a características de hardware que ponen al algoritmo serial en desventaja. Por ejemplo, los datos para un problema podrían ser demasiado grandes para caber en la memoria principal de un solo procesador, degradando su performance debido al uso de almacenamiento

secundario. Pero cuando se divide entre varios procesadores, las particiones de datos individuales serían lo suficientemente chicas para caber en las respectivas memorias de los procesadores. Con frecuencia, la obtención de speedup superlineal es una indicación de que el código secuencial tiene problemas de *cache miss*.

En la literatura pueden encontrarse numerosos comentarios respecto que el speedup no puede exceder al número de procesadores $p$. Esto puede ser "probado" mostrando cómo un único procesador puede simular cualquier algoritmo paralelo que corre en tiempo $t(I)$ en un solo procesador en tiempo $p\,t(I)$ (utilizando *round robin*). Luego, existe un algoritmo serial con tiempo de ejecución $p\,t(I)$. Por lo tanto, el mejor algoritmo secuencial para el problema no toma más que ese tiempo, y el speedup real no es más que $p$. Este argumento presenta, al menos, dos defectos:

1. Existen problemas para los cuales no hay algoritmos secuenciales que sean "mejores" en todas las instancias. Por esto, el speedup real es computado en relación al algoritmo que sería usado en la práctica. Con respecto a este algoritmo, es posible que otros secuenciales sean más rápidos en ciertas instancias y más lentos en otras. La versión paralela de éstos podría entonces dar speedup mayor que $p$ en algunas instancias y speedup menor que 1 en otras.

2. La simulación del algoritmo paralelo por un solo procesador incurre en algún overhead. Por ejemplo, la máquina paralela puede tener más memoria agregada y entonces la simulación puede necesitar almacenamiento secundario, la frecuencia de éxitos en la cache puede decrecer en la simulación ya que el tamaño del cache del procesador es solo igual a la de uno de los del ensamble, y además existe un costo de ciclado entre los programas en *round robin*.

Como resultado, el uniprocesador en realidad toma tiempo $c\,p\,t(I)$ para algún $c > 1$. Luego, el speedup entre el tiempo del procesador único y el tiempo paralelo es $c\,p\,t(I) - t(I) > p$.

Algunos artículos clásicos en los que se pueden encontrar observaciones analíticas y/o experimentales de speedup superlineal son [5, 6, 160, 216, 259, 286, 325, 340]

En [147], Gustafson expresa que el speedup superlineal que resulta de ineficiencia en el algoritmo serial es "efímero y poco interesante", y que existen otras fuentes de speedup superlineal, como las diferentes velocidades de memoria inherentes en ensambles de memoria distribuida y el *shift* en la fracción de tiempo gastada en tareas de distinta velocidad.

## 4.5 Overhead paralelo

El *overhead paralelo total* $T_o$ es la suma de los overheads en que incurren todos los procesadores debido al procesamiento paralelo. Incluye costos de comunicación, trabajo no esencial y tiempo ocioso debido a sincronización y componentes seriales del algoritmo. Matemáticamente, la *función de overhead* está dada por

$$T_o = p\,T_p - T_s \tag{4.2}$$

Asumiendo (para simplificar) que $T_o$ es una cantidad no negativa, el speedup está acotado por $p$. Por ejemplo, el speedup puede ser superlineal y $T_o$ negativo si la memoria es jerárquica y el tiempo de acceso crece a medida que aumenta la memoria utilizada por el programa. En este caso, la velocidad de computación efectiva de un programa *grande* será menor sobre un procesador serial que en una máquina paralela con procesadores similares. La razón es que un algoritmo secuencial que usa $M$ bytes de memoria usará sólo $M/p$ bytes en cada procesador de una máquina $p$-procesador. La suposición de memoria *flat* ayuda a concentrarse en las características del algoritmo y arquitectura paralela, sin entrar en detalles de una máquina particular.

Para un sistema paralelo dado, $T_o$ normalmente es una función de $W$ y $p$, por lo que suele denotarse $T_o(W, p)$.

## 4.6 Eficiencia

La *eficiencia* ($E$) es una medida de performance paralela estrechamente relacionada con el speedup. Está dada por el cociente entre el speedup ($S$) y el número de procesadores ($p$):

$$E = \frac{S}{p} = \frac{T_s}{p\,T_p} = \frac{1}{1 + \frac{T_o}{T_s}} \tag{4.3}$$

Puede pensarse en la eficiencia como el speedup promedio por procesador. Los procesadores no brindan el 100 por ciento de su tiempo para cómputo, de modo que la eficiencia mide la fracción de tiempo que son *útiles*.

El valor de la eficiencia está entre 0 y 1, dependiendo del grado de efectividad con el cual se utilizan los procesadores. Cuando es 1, corresponde al speedup perfecto.

**Ejemplo 4.2** *Eficiencia de la suma n números en un hipercubo n-procesador.*

La eficiencia para sumar $n$ números en un hipercubo $n$-procesador (planteado en el Ejemplo 4.1) es $E = \Theta(1/\log n)$ [209]. □

Dependiendo de la variedad de speedup usada, puede obtenerse una variedad diferente de eficiencia. Dado que el speedup puede exceder $p$ (superlineal), la eficiencia podría exceder 1. Ambos factores no deberían ser usados como una métrica de performance independiente del tiempo de corrida [292].

En la literatura pueden encontrarse formulaciones alternativas de eficiencia. Por ejemplo, en [60] Carmona y Rice la definen como el cociente del trabajo realizado por el algoritmo paralelo ($wa$) y el trabajo consumido por el algoritmo ($we$). Si se define el trabajo realizado por el algoritmo paralelo como el que habría sido hecho por el "mejor" algoritmo secuencial, y el trabajo consumido como el producto del tiempo de ejecución paralelo, la velocidad ($V$) de un procesador paralelo individual, y el número ($p$) de procesadores, y se asume que todos los procesadores tienen la misma velocidad,

$$we = (tiempo\,paralelo)\,V\,p$$

$$wa = (mejor\ tiempo\ secuencial)\,V$$

$$\frac{wa}{we} = \frac{mejor\ tiempo\ secuencial}{p\,(tiempo\,paralelo)}$$

lo que equivale al speedup real dividido por $p$. De esta forma, $wa/we$ equivale a la eficiencia real. Análogamente, si $wa$ está definido como el trabajo hecho por el algoritmo paralelo cuando es ejecutado sobre un solo procesador, entonces $wa/we$ equivale a la eficiencia relativa (esto es, speedup relativo dividido por $p$).

Otra definición alternativa de eficiencia se da en [60]. Definiendo *trabajo desperdiciado* ($ww$) como $we - wa$, la eficiencia puede escribirse como

$$\frac{wa}{we} = \frac{wa}{ww + wa} = \frac{1}{1 + \frac{ww}{wa}}$$

## 4.7 Costo

El *costo* de un sistema paralelo se define como el producto del tiempo de ejecución paralelo ($T_p$) y el número de procesadores utilizados ($p$). Refleja la suma del tiempo que cada procesador utiliza resolviendo el problema.

Puede expresarse la eficiencia como el cociente entre el tiempo de ejecución del algoritmo secuencial más rápido conocido para resolver un problema y el costo de resolver el mismo problema en $p$ procesadores.

Se dice que el sistema paralelo es de *costo óptimo* si y solo si el costo es asintóticamente del mismo orden de magnitud que el tiempo de ejecución serial, es decir $pT_p = \Theta(W)$. Esto es, el costo de resolver un problema en una máquina paralela es proporcional al tiempo de ejecución del algoritmo secuencial conocido más rápido en un solo procesador. Dado que la eficiencia es el cociente entre el costo secuencial y el costo paralelo, un sistema paralelo de costo óptimo tiene una eficiencia de $\Theta(1)$.

También suele encontrarse referenciado como *trabajo* o *producto procesador-tiempo*, y un sistema de costo óptimo es conocido como un sistema $pT_p$-óptimo.

**Ejemplo 4.3**

El costo de sumar $n$ números en un hipercubo $n$-procesador mostrado en el Ejemplo 4.1 es de $\Theta(n \log n)$. Dado que el tiempo de ejecución serial es $\Theta(n)$, el sistema paralelo no es de costo óptimo. Esto mismo se observa a partir del Ejemplo 4.2 en que se muestra que la eficiencia en este caso es menor que $\Theta(1)$ [209]. □

## 4.8 Grado de concurrencia

El *grado de concurrencia* o *grado de paralelismo* ($C(W)$) es el número máximo de tareas que pueden ser ejecutadas simultáneamente en cualquier momento en el algoritmo paralelo. Para un $W$ dado, el algoritmo paralelo no puede usar más de $C(W)$ procesadores. $C(W)$ depende sólo del algoritmo paralelo, y es independiente de la arquitectura. Es una función discreta del tiempo, y refleja cómo el paralelismo de software *matchea* con el de hardware. La gráfica del grado de concurrencia como función del tiempo se denomina *perfil de concurrencia* o *perfil de paralelismo* de un programa.

Por ejemplo, para multiplicar dos matrices $n \times n$ usando el algoritmo de multiplicación paralelo de Fox [113], $W = n^3$ y $C(W) = n^2 = W^{2/3}$. En este caso el producto procesador-tiempo es $\Theta(W)$, esto es, el algoritmo es de costo óptimo y entonces $C(W) \leq \Theta(W)$.

Así definido, el grado de concurrencia supone un número ilimitado de procesadores y otros recursos necesarios disponibles, aunque esto no siempre puede ser alcanzable en una computadora real con recursos limitados.

## 4.9 Efecto de la granularidad y el mapeo sobre la performance

Los ejemplos muestran un algoritmo que no es de costo óptimo. Básicamente, el problema es que se utilizan tantos procesadores como número de entradas, lo cual es excesivo. En la práctica, pueden asignarse porciones más grandes de entrada a cada procesador. Esto corresponde a incrementar la *granularidad* de la computación. La técnica de usar menos procesadores que el máximo posible para ejecutar un algoritmo paralelo se denomina *scaling down* (reducción progresiva) del sistema paralelo en términos del número de procesadores [209, 141].

Una forma de reducir un sistema paralelo es diseñar un algoritmo para un elemento de entrada por procesador, y luego usar menos procesadores para simular una cantidad mayor. Si hay $n$ entradas y $p$ procesadores ($p < n$), se puede usar el algoritmo diseñado asumiendo $n$ procesadores virtuales y haciendo que cada uno de los $p$ procesadores físicos simule $n/p$ virtuales. Como el número de procesadores decrece por un factor de $n/p$, la computación en cada uno crece por el mismo factor, ya que cada uno ahora realiza el trabajo de $n/p$. Si los virtuales están mapeados apropiadamente en los físicos, el tiempo de comunicación general no crece más que un factor de $n/p$. El tiempo de ejecución paralelo total crece, a lo sumo, por un factor de $n/p$, y el producto procesador-tiempo no crece. Por lo tanto, si un sistema paralelo con $n$ procesadores es de costo óptimo, usar $p$ procesadores para simular $n$ preserva esta optimalidad.

Una desventaja del método de crecer la granularidad computacional es que si un sistema paralelo no es de costo óptimo inicialmente, puede aún no serlo después del incremento de granularidad. Esto se muestra en el siguiente ejemplo.

**Ejemplo 4.4** *Suma de $n$ números en un hipercubo $p$-procesador.*

Sea el problema de sumar $n$ números en $p$ procesadores ($p < n$, ambos potencia de 2). Puede usarse el mismo algoritmo que en el Ejemplo 4.1, simulando $n$ procesadores en $p$, como muestra la Figura 4.2

El procesador virtual $i$ es simulado por el físico con label $i \oplus p$; los números a ser sumados son distribuidos de manera similar. Los primeros $\log p$ de los $\log n$ pasos del algoritmo original son simulados en $((n/p)\log p)$ pasos en $p$ procesadores. En los pasos restantes, no se requiere comunicación porque los procesadores que se comunican en el algoritmo original son simulados por el mismo; por lo tanto, los números restantes son sumados localmente.

El algoritmo toma $\Theta((n/p)\log p)$ tiempo en los pasos que requiere comunicación, y luego un solo procesador se queda con $n/p$ números para sumar, tomando tiempo $\Theta(n/p)$. Así, el tiempo de ejecución paralela es $\Theta((n/p)\log p)$, y el costo es $\Theta(n \log p)$

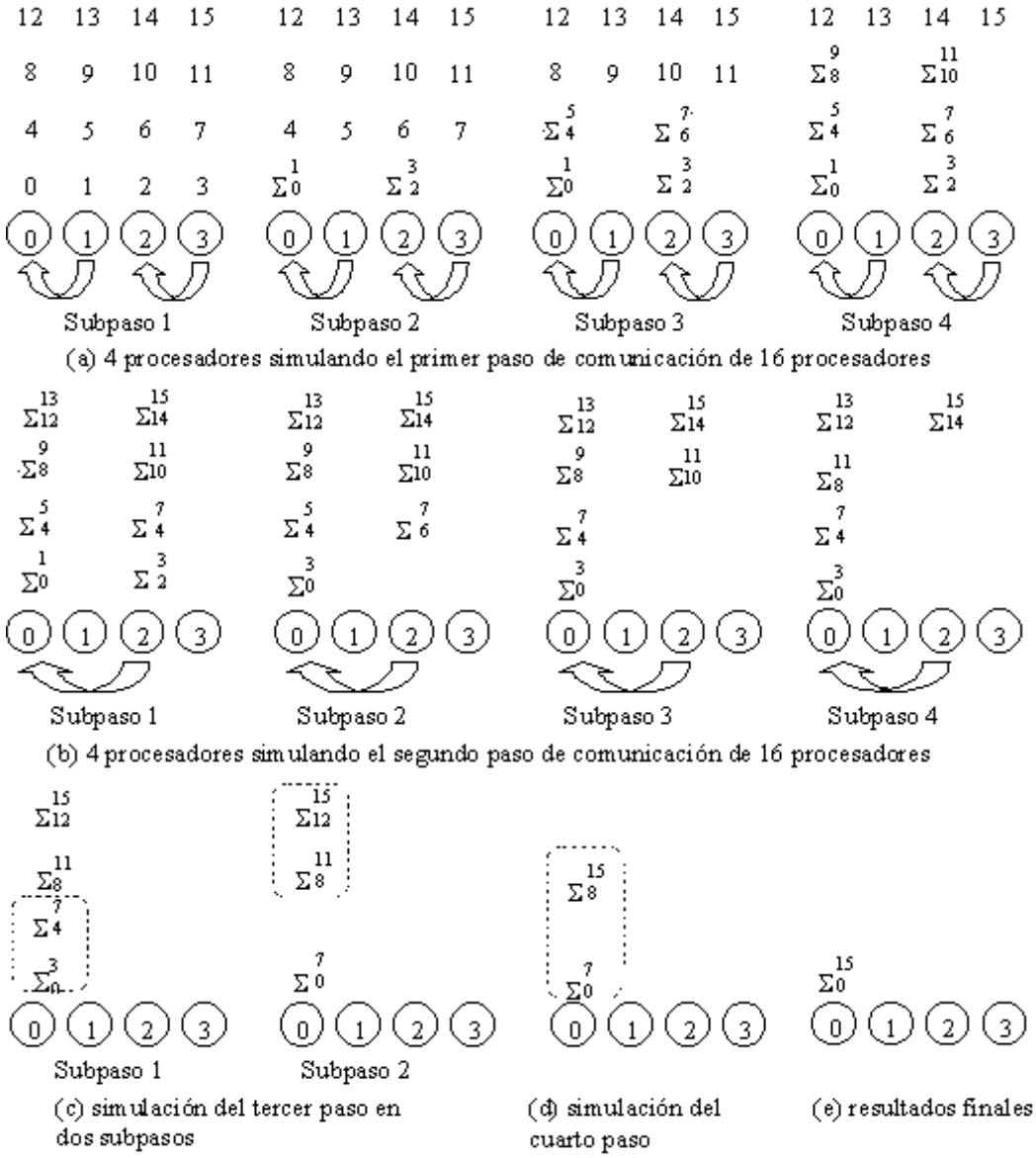


Figura 4.2: 4 procesadores simulando 16 procesadores para computar la suma de 16 números

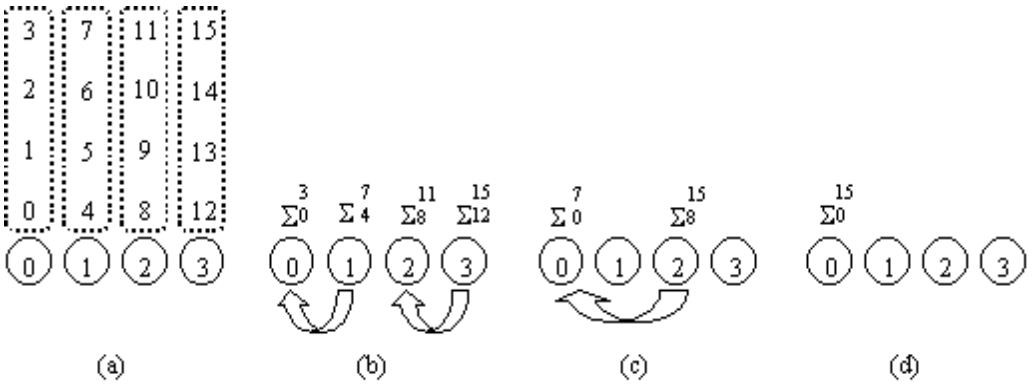


Figura 4.3: Suma de 16 números en un hipercubo 4-procesador, con costo óptimo

(asintóticamente mayor que el costo $\Theta(n)$ de sumar $n$ números secuencialmente). Por lo tanto, el sistema paralelo no es de costo óptimo [209]. □

El Ejemplo 4.1 mostró que pueden sumarse $n$ números en un hipercubo $n$-procesador en tiempo $\Theta(\log n)$. Cuando se usan $p$ procesadores para simular $n$, el tiempo de ejecución paralelo esperado es $\Theta((n/p)\log n)$. Pero, en el Ejemplo 4.4 la tarea se realizó en $\Theta((n/p)\log p)$. La razón es que cada paso de comunicación del algoritmo original no tiene que ser simulado; a veces, la comunicación tiene lugar entre procesadores virtuales simulados por el mismo procesador físico. Por ejemplo, la simulación de los pasos 3 y 4 no requiere comunicación. Pero, esta reducción en comunicación no fue suficiente para hacer al algoritmo de costo óptimo. El siguiente Ejemplo muestra que el problema puede ser resuelto con costo óptimo con una asignación más inteligente de datos a procesadores.

**Ejemplo 4.5** *Suma de n números con costo óptimo en un hipercubo.*

La Figura 4.3 muestra un método alternativo para sumar $n$ números usando $p$ procesadores. En el primer paso, cada procesador suma localmente sus $n/p$ números en tiempo $\Theta(n/p)$. Ahora el problema se reduce a sumar las $p$ sumas parciales en $p$ procesadores, lo que puede hacerse en tiempo $\Theta(\log p)$ por el método descripto en el Ejemplo 4.1. El tiempo de ejecución paralelo de ese algoritmo es $\Theta(n/p+\log p)$ y su costo es $\Theta(n+p\log p)$. Cuando $n = \Omega(p\log p)$, el costo es $\Theta(n)$, que es igual al tiempo de ejecución. Por lo tanto, el sistema paralelo es de costo óptimo [209]. □

Los ejemplos muestran que la manera en la cual la computación se mapea en los procesadores puede determinar si un sistema paralelo es de costo óptimo. Sin embargo, no puede hacerse de costo óptimo a todos los sistemas que no lo son haciendo *scaling down* del número de procesadores.

Con un mapeo de datos adecuado, usar menos procesadores físicos para simular un número mayor de procesadores virtuales mantiene la optimalidad en costo si el sistema

paralelo es de costo óptimo inicialmente. Pero, si el sistema paralelo no es de costo óptimo para un gran número de procesadores, entonces una simulación no resulta necesariamente en un sistema de costo óptimo. Aún en el primer caso, más allá de preservar la optimalidad, un enfoque ingenuo puede resultar en una formulación inferior en términos de tiempo de ejecución paralelo.

Una simulación de varios procesadores por menos puede no tener en cuenta que hay múltiples maneras de asignar $n$ procesadores virtuales a $p$ físicos ($n > p$). La performance puede ser diferente para distintas asignaciones. En el Ejemplo 4.1, la tarea de dividir $n$ entradas entre $n$ procesadores era trivial pues a cada procesador se le asignó un solo elemento. Pero, si las $n$ entradas son mapeadas a $p$ procesadores, donde $n > p$, entonces hay más de una manera de hacerlo. Como muestran los Ejemplos 4.4 y 4.5, el tiempo paralelo del mismo problema es una función del mapeo de procesadores virtuales en físicos.

Un algoritmo que requiere $W$ pasos de computación básicos puede ser mapeado en un máximo de $W$ procesadores, y cada procesador realiza un solo paso del algoritmo secuencial. Si se usan menos procesadores, entonces cada uno resuelve una parte mayor del problema entero. Puede haber distintos algoritmos secuenciales para resolver una parte del problema localmente en un procesador. A veces, la elección del método para realizar la computación local afecta el tiempo de ejecución paralelo asintótico. De hecho, la elección del mejor algoritmo para realizar las computaciones locales en cada procesador puede depender del número de procesadores. Ejemplos de tales sistemas paralelos incluyen algunos algoritmos de sorting y multiplicación de matrices.

El algoritmo paralelo óptimo para resolver un problema sobre un número arbitrario de procesadores no puede ser obtenido trivialmente desde el algoritmo paralelo de grano más fino. Más aún: un análisis del sistema paralelo basado en la formulación de grano más fino puede ocultar el efecto de ciertos rasgos de hardware sobre la performance del sistema paralelo. Por ejemplo, el tiempo de transferencia de un mensaje entre dos procesadores es el mismo con ruteos *store-and-forward* y *cut-through* si el mensaje contiene sólo una palabra. Por el contrario, el ruteo *cut-through* con frecuencia permite transferencias mucho más rápidas de mensajes grandes que el ruteo *store-and-forward*. La performance de varios algoritmos paralelos es casi idéntica en un hipercubo y una mesh con ruteo *cut-through*; sin embargo, su performance es mucho peor en una mesh con *store-and-forward*. Un análisis del algoritmo paralelo de grano más fino puede no revelar estos hechos importantes.

Esta discusión ilustra que diseñar un algoritmo paralelo eficiente involucra más que desarrollar un algoritmo para un elemento de entrada o para una computación por procesador. Concebir el algoritmo de grano más fino usualmente es fácil y puede servir como primer paso lógico hacia el desarrollo de un algoritmo paralelo. Pero, el diseño completo debería tener en cuenta el mapeo de datos en procesadores e incluir una descripción de su implementación en un número arbitrario de PEs. Por eso es común tomar el tamaño de la entrada y el número de procesadores como dos variables separadas al diseñar y analizar algoritmos paralelos.

## 4.10 Otras medidas

En [223] Lee definió varios parámetros para evaluar las computaciones paralelas; en muchos casos de aplicaciones reales se encuentran tradeoffs de estos factores.

Sea $Op(p)$ el número total de operaciones unitarias realizadas por un sistema $p$-procesador y $T(p)$ el tiempo de ejecución en pasos de tiempo unitarios. En general, $T(p) < Op(p)$ si más de una operación es realizada por $p$ procesadores por unidad de tiempo, donde $p \geq 2$. Se asume $T(1) = O(1)$ en un sistema uniprocesador.

La *redundancia* en una computación paralela está definida como

$$R(p) = \frac{Op(p)}{Op(1)}$$

y representa la extensión del *matching* entre paralelismo de software y de hardware. Además, $1 \leq R(p) \leq p$.

La *utilización del sistema* en una computación paralela se define como

$$U(p) = R(p) * E(p) = \frac{Op(p)}{p\,T(p)}$$

(donde $E(p)$ es la eficiencia con $p$ procesadores) e indica el porcentaje de recursos (procesadores, memoria, etc.) que estuvieron ocupados durante la ejecución de un programa paralelo. Notar que $1/p \leq E(p) \leq U(p) \leq 1$ y $1 \leq R(p) \leq 1/E/p) \leq n$.

La *calidad* de una computación paralela es directamente proporcional al speedup y la eficiencia e inversamente relacionada con la redundancia. Así,

$$Q(p) = \frac{S(p) * E(p)}{R(p)} = \frac{T^3(1)}{p\,T^2(p)\,Op(p)}.$$

Dado que $E(p)$ es una fracción y $R(p)$ un número entre 1 y $p$, la calidad $Q(p)$ está acotada superiormente por el speedup $S(p)$.

La redundancia mide la extensión del crecimiento de la carga de trabajo. La utilización indica cuánto son usados los recursos durante una computación paralela. Finalmente, la calidad combina los efectos de speedup, eficiencia y redundancia en una única expresión para dar el mérito relativo de una computación paralela sobre un sistema de cómputo.

## 4.11 Cargas de trabajo y Modelos de speedup

En general, si la carga de trabajo o tamaño del problema se mantiene sin cambios (carga *constante*), entonces la eficiencia $E$ decrece rápidamente cuando el tamaño de la máquina paralela (cantidad de procesadores, $p$) crece. La razón es que el overhead crece más rápido que $p$. Intuitivamente, para mantener la eficiencia a un nivel deseado es necesario incrementar ambos tamaños (el de la máquina y el del problema) proporcionalmente. Esto sistema es conocido como *computadora escalable* para resolver *problemas escalados*.

En el caso ideal, la carga de trabajo crece como una función líneal de $p$ (escalabilidad lineal). Si esta curva no es alcanzable, una posibilidad es obtener una escalabilidad sublineal tan cercana a la linealidad como sea posible. Si en cambio la carga sigue un patrón de crecimiento exponencial volviéndose extremadamente grande, el sistema es considerado poco escalable; esto se debe a que para mantener buena eficiencia o speedup es necesario un incremento explosivo del tamaño del problema, y podría exceder los límites de memoria o entrada/salida.

Los tres modelos de performance más conocidos son el que se basa en una carga de trabajo o un tamaño de problema fijo (Amdahl, 1967), el aplicable a problemas escalados donde el tamaño de éste crece con el de la máquina (Gustafson, 1987), y el modelo de speedup para problemas escalados limitados por capacidad de memoria (Sun y Ni, 1993).

### 4.11.1 Modelo de carga fija o tamaño fijo

En varias aplicaciones prácticas (como las que demandan respuesta en tiempo real), la carga de trabajo computacional suele establecerse con un tamaño de problema fijo. A medida que $p$ crece, la carga de trabajo fija se distribuye en más procesadores para su ejecución paralela. Por lo tanto, el objetivo principal es producir los resultados tan rápido como sea posible. En otras palabras, el objetivo es tiempo mínimo. El speedup obtenido para aplicaciones de tiempo crítico se llama *speedup de carga fija*.

Las formulaciones tradicionales de speedup, incluyendo la *ley de Amdahl* [14, 354], están basadas en un tamaño de problema fijo y así en una carga fija. El factor de speedup está acotado superiormente por un cuello de botella secuencial. Intuitivamente, si se considera que el algoritmo se compone de una parte serial (no paralelizable) y una porción que puede ser paralelizada, el máximo speedup alcanzable siempre estará acotado por la componente secuencial.

Como se expresa en [182], puede definirse el factor de speedup de carga fija como:

$$S_p = \frac{T_1}{T_p} = \frac{\sum_{i=1}^{q} W_i}{\sum_{i=1}^{q} (\frac{W_i}{i} \lceil \frac{i}{p} \rceil)} \tag{4.4}$$

donde $T_1$ y $T_p$ son los tiempos de respuesta con 1 y $p$ procesadores, $q$ es el máximo grado de paralelismo, y $W_i$ es el trabajo realizado con grado de paralelismo $i$.

Hay una serie de factores que pueden degradar la performance de speedup, incluyendo latencias de comunicación, overhead de sistema operativo causado por interrupciones, etc. Todos estos overheads se agregarían como un término aditivo $Q(p)$ al denominador de la Ec. 4.4. Para simplificar, en lo que sigue, se asume 0 dicho término.

En 1967, Gene Amdahl derivó un speedup de carga fija para el caso especial en que la máquina opera sólo en modo secuencial (grado de paralelismo igual a 1) o en modo perfectamente paralelo (grado de paralelismo $p$). En este caso, la Ec. 4.4 se simplifica a

$$S_p = \frac{W_1 + W_p}{W_1 + W_p/p} \tag{4.5}$$

Esto significa que la porción secuencial del programa no cambia con respecto al tamaño de la máquina, pero el segmento paralelo es ejecutado en forma equitativa por los $p$ procesadores resultando en un tiempo reducido. Para una situación normalizada en que $W_1 + W_p = \alpha + (1 - \alpha) = 1$, con $\alpha = W_1$ y $(1 - \alpha) = W_p$ se tiene

$$S_p = \frac{p}{1 + (p - 1)\alpha} \tag{4.6}$$

donde $\alpha$ representa el porcentaje del programa que debe ser ejecutado secuencialmente y $1 - \alpha$ la porción que puede ejecutarse en paralelo.

La *ley de Amdahl* se ilustra en la Figura 4.4 [182]. Cuando el número de procesadores crece, la carga de cada uno es menor. La carga total se mantiene constante y el tiempo total de ejecución decrece. Eventualmente, la parte secuencial dominará la performance. El speedup máximo será $p$ cuando $\alpha = 0$, y el mínimo será 1 cuando $\alpha = 1$. Cuando $p \to \infty$, el valor límite de $S_p \to 1/\alpha$. Esto implica que el speedup está acotado superiormente por $1/\alpha$ cuando el tamaño de la máquina se vuelve muy grande.

La curva de speedup cae muy rápidamente cuando $\alpha$ crece, por lo que con un pequeño porcentaje de código secuencial la performance completa no puede ir más allá de $1/\alpha$ (*cuello de botella secuencial*). Este argumento impuso durante dos décadas una visión muy pesimista respecto del procesamiento paralelo.

La tendencia de la medida tradicional de speedup es a favorecer a los procesadores más lentos y a los programas peor codificados. Esto fue observado por Barton y Withers [31], quienes estudiaron el speedup para 4 versiones del Intel iPSC sobre el mismo algoritmo paralelo: cuanto más rápido era el procesador, menor era el speedup por usar más procesadores.

Probablemente una razón por la que muchas veces no se reconoce la naturaleza bidi-

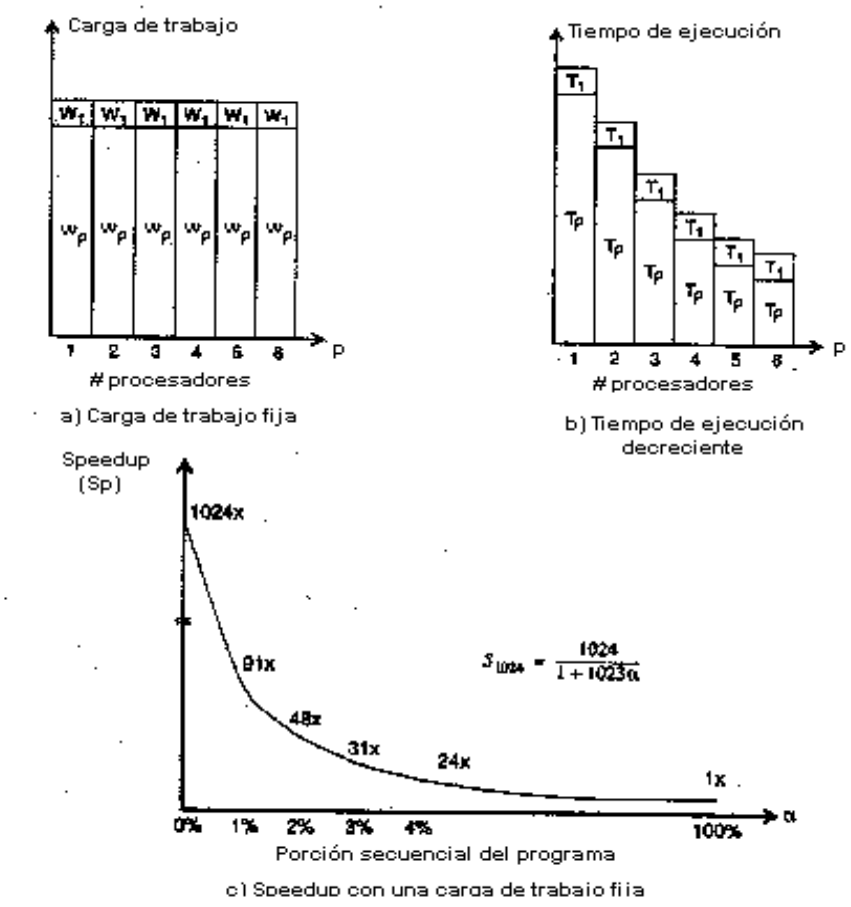


Figura 4.4: Modelo de speedup de carga de trabajo fija y ley de Amdahl

mensional de los datos de speedup (dependencia sobre el número de procesadores y el tamaño del problema) sea el deseo de medir el speedup de programas complicados estrictamente por experimento, debido a la dificultad de encontrar una expresión analítica de la complejidad. De ahí que la gráfica clásica de speedup sea una curva, o a lo sumo un conjunto de curvas para distintos tamaños de problema (aunque cada curva solo cubre un rango parcial de valores de $p$). Esto da una forma de speedup escalado aunque un tanto "oscura" debido al conjunto de curvas.

### 4.11.2 Modelo de tiempo fijo

Uno de los principales "defectos" al aplicar la ley de Amdahl es que el tamaño del problema (carga de trabajo) se mantiene fijo y no puede escalarse para adecuarlo al poder de cómputo disponible cuando el tamaño de máquina crece. En otras palabras, la carga fija previene la escalabilidad en performance. Aunque el cuello de botella secuencial es un problema serio, puede aliviarse removiendo la restricción de carga fija (o tamaño de problema fijo). Esto fue observado por Gustafson en [146], donde propuso el concepto de *speedup de tiempo fijo*.

Las aplicaciones de tiempo crítico promovieron el desarrollo del modelo de speedup de carga fija y la ley de Amdahl. Existen otras aplicaciones que enfatizan la precisión más que el tiempo mínimo. Cuando el tamaño de máquina es modificado para obtener mayor poder de cómputo, puede quererse incrementar el tamaño del problema para crear una mayor carga de trabajo, produciendo una solución más precisa y manteniendo el tiempo de ejecución sin cambios.

En el sentido usual, *speedup escalado* significa que el tamaño del problema crece con la cantidad de procesadores. La pregunta es ¿cuánto crece?. Si $N_0$ es el tamaño del problema que cabe en la memoria local de un único procesador, entonces el tamaño de problema es generalmente escalado para ajustarse a la memoria total de los $p$ procesadores. Si el almacenamiento es proporcional a $N_0$, entonces $p$ procesadores corren un problema de tamaño $p\,N_0$. Luego, el speedup escalado está dado por el cociente entre la complejidad del mejor algoritmo serial para un problema de tamaño $p\,N_0$ y la complejidad del algoritmo paralelo para un problema del mismo tamaño en $p$ procesadores. Esta expresión podría requerir una aproximación de memoria plana debido a que el denominador podría ser demasiado grande para un único procesador del ensamble. Entre los primeros "usuarios" de speedup escalado se encontraron Seitz, Gustafson y Moler.

La idea de que el tiempo debería fijarse cuando se realiza evaluación de performance de problemas escalables fue presentada por primera vez en [146], aunque Worley ya lo había notado en la resolución de ecuaciones diferenciales parciales sobre ensambles de computadoras. La distinción entre speedup escalado y *speedup de tiempo fijo* apareció alrededor de 1988. En algunos casos se utiliza el término "speedup escalado" para sig-

nificar cualquier medida que permita que el tamaño del problema cambie. En el modelo de tiempo fijo, es el *trabajo* el que crece con $p$, no el almacenamiento.

Muchas modelizaciones científicas y aplicaciones de simulación demandan la solución de problemas de matrices a gran escala basados en formulaciones de ecuaciones diferenciales parciales (PDE) discretizadas con un gran número de puntos de grilla. Ejemplos representativos incluyen el uso de métodos de elementos finitos para realizar análisis estructural, o el uso de métodos de diferencias finitas para resolver problemas computacionales de dinámica de fluidos en pronóstico meteorológico.

Las grillas gruesas requieren menos computación; las finas necesitan muchas más, brindando mayor precisión. La simulación de pronóstico meteorológico puede demandar la solución de PDEs de 4 dimensiones. Si se reduce el espaciado de grilla en cada dimensión física ($x$, $y$, y $z$) en un factor de 10, y se incrementan los pasos de tiempo en la misma magnitud, entonces habrá un incremento de $10^4$ veces más puntos de grilla. Luego, la carga de trabajo será al menos 10000 veces mayor; tal escalado de problema necesita mayor poder de cómputo para obtener el mismo tiempo de ejecución.

La principal ventaja en este caso no es ahorrar tiempo sino producir resultados mucho más precisos. Este escalado de problema para precisión motivó a Gustafson desarrollar un modelo de speedup de tiempo fijo. El problema escalado mantiene todos los recursos ocupados, resultando en un mejor cociente de utilización del sistema.

En las aplicaciones de precisión crítica, se quiere resolver el tamaño de problema más grande posible sobre una máquina más grande con aproximadamente el mismo tiempo de ejecución que para resolver un problema más chico sobre una máquina más chica. Cuando el tamaño de máquina crece, hay que tratar con una carga de trabajo incrementada y un nuevo perfil de paralelismo. Sea $q'$ el máximo grado de paralelismo con respecto al problema escalado y $W_i'$ la carga de trabajo escalada con grado de paralelismo $i$.

En general $W_i' > W_i$ para $2 \leq i \leq q'$ y $W_1' = W_1$. El speedup de tiempo fijo se define bajo la suposición de que $T_1 = T_p'$, donde $T_p'$ es el tiempo de ejecución del problema escalado y $T_1$ corresponde al problema original sin escalar. Una fórmula general para speedup de tiempo fijo se define por $S_p' = T_1/T_p'$, modificado de la Ec. 4.4:

$$S_p' = \frac{T_1}{T_p'} = \frac{\sum_{i=1}^{q'} W_i'}{\sum_{i=1}^{q'} \frac{W_i'}{i}\lceil \frac{i}{p} \rceil + Q(p)} = \frac{\sum_{i=1}^{q'} W_i'}{\sum_{i=1}^{q} W_i} \tag{4.7}$$

El speedup de tiempo fijo fue desarrollado originalmente por Gustafson para un perfil de paralelismo especial con $W_i = 0$ si $i \neq 1$ e $i \neq p$. De manera similar a la ley de Amdahl, se puede reescribir la Ec. 4.7 como sigue, asumiendo $Q(p) = 0$:

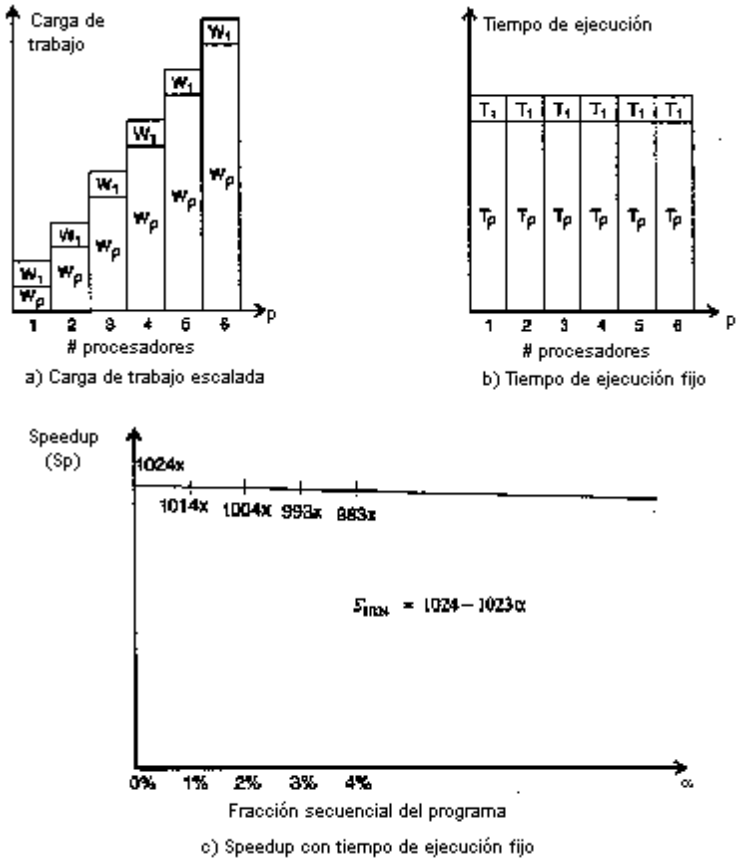


Figura 4.5: Modelo de speedup de tiempo fijo y ley de Gustafson

$$S'_p = \frac{\sum_{i=1}^{q'} W'_i}{\sum_{i=1}^{q} W_i} = \frac{W'_1 + W'_p}{W_1 + W_p} = \frac{W_1 + p\,W_p}{W_1 + W_p} \tag{4.8}$$

donde $W'_p = p\,W_p$ y $W_1 + W_p = W'_1 + W'_p/p$, correspondiendo a la condición de tiempo fijo. De la Ec. 4.8, la carga de trabajo paralela $W'_p$ fue escalada hasta $p$ veces $W_p$ de manera lineal.

La relación de una carga de trabajo escalada con el speedup escalado de Gustafson se muestra en la Figura 4.5 [182]. De hecho, la *ley de Gustafson* puede ser reescrita en términos de $\alpha = W_1$ y $1 - \alpha = W_p$ bajo la misma suposición hecha para la ley de Amdahl de que $W_1 + W_p = 1$:

$$S'_p = \frac{\alpha + p(1-\alpha)}{\alpha + (1-\alpha)} = p - \alpha(p-1) \tag{4.9}$$

En la Figura 4.5(a), se muestra la situación al escalar la carga de trabajo. La Figura 4.5(b) muestra el estilo de ejecución de tiempo fijo. La Figura 4.5(c) plotea $S'_p$ como una función de la porción secuencial $\alpha$ de un programa corriendo en un sistema con $p = 1024$ procesadores [145, 149, 230]; la pendiente de la curva en este caso es mucho más chata que para carga fija. Esto implica que la ley de Gustafson soporta performance escalable cuando crece el tamaño de la máquina. La idea es mantener a todos los procesadores ocupados incrementando el tamaño del problema. Cuando el problema puede escalar para matchear el poder de cómputo disponible, la fracción secuencial ya no es un cuello de botella y puede extraerse eficiencia muy alta de un procesador masivamente paralelo.

En [149], Gustafson enumeró una serie de consecuencias del modelo de tiempo fijo. Por ejemplo: que el speedup de tiempo fijo no favorece a los procesadores más lentos, que la medición de tiempo fijo crea un nuevo tipo de speedup superlineal, y que el tiempo fijo predice nuevos límites al speedup paralelo.

En [308], Yuan Shi plantea que las leyes de Amdahl y Gustafson son idénticas, y que en realidad son dos *formulaciones* distintas de *una* ley. Su mayor justificación es que los porcentajes seriales de cada caso no son iguales sino que están relacionados por una ecuación, y que al reemplazar el porcentaje serial dependiente del número de procesadores en la formulación de Gustafson se obtiene una fórmula idéntica a la de Amdahl. Además, indica que un prerrequisito para aplicar la ley de Amdahl es que el algoritmo secuencial retenga su estructura tal que el *mismo número* de instrucciones sean procesadas por las implementaciones serial y paralela para la misma entrada; ésto no siempre es posible cuando los programas son particionados, y puede derivar en un "abuso" de la ley. Shi sugiere usar métodos basados en el tiempo de procesamiento para la evaluación de performance paralela.

### 4.11.3 Modelo de memoria limitada

Tanto la ley de Amdahl como el modelo de speedup de Gustafson utilizan un único parámetro (la porción secuencial de un algoritmo paralelo) para caracterizar una aplicación. Ambos son simples y brindan una idea de la degradación potencial del paralelismo al incrementar la cantidad de procesadores. La ley de Amdahl trata con un tamaño de problema fijo y busca cuan chico puede ser el tiempo de respuesta, sugiriendo que el paralelismo no puede alcanzar speedup alto. Gustafson fija el tiempo de respuesta y se interesa en cuál es el problema más grande que puede resolverse en ese tiempo.

En [337] Sun y Ni investigan en profundidad la escalabilidad de los problemas. Mientras los problemas escalables de Gustafson están restringidos por el tiempo de ejecución, la capacidad de memoria principal es también una métrica crítica. Para computadoras paralelas, especialmente multiprocesadores de memoria distribuida, el tamaño de problemas escalables con frecuencia está determinado por la memoria disponible. La poca

memoria se paga con tiempo de solución del problema (debido a las demoras de E/S y pasaje de mensajes) y en tiempo del programador (debido al código adicional requerido para multiplexar la memoria distribuida). Para muchas aplicaciones, el almacenamiento es una restricción importante para escalar el tamaño de problema.

El modelo de *speedup de memoria limitada* de Sun y Ni generaliza las leyes de Amdahl y Gustafson para maximizar el uso de CPU y memoria. La idea es resolver el problema más grande posible, limitado por el espacio de memoria. Esto también demanda una carga de trabajo escalada, proveyendo mayor speedup, precisión y uso de recursos.

Con frecuencia las computaciones científicas o ingenieriles de gran escala requieren espacio mayor en memoria. De hecho, varias aplicaciones de máquinas paralelas son de memoria limitada más que de CPU o E/S limitada. Esto es especialmente verdad en un sistema multicomputador con memoria distribuida: la memoria local de cada nodo es relativamente chica. Por lo tanto, cada nodo puede manejar sólo un subproblema pequeño.

Cuando un gran número de nodos son usados colectivamente para resolver un único gran problema, la capacidad de memoria total crece de manera proporcional. Esto permite al sistema resolver un problema escalado particionando o replicando el programa y descomponiendo el dominio del conjunto de datos. En lugar de mantener fijo el tiempo de ejecución, uno puede querer usar toda la memoria incrementada escalando más el tamaño del problema. En otras palabras: si existe un espacio de memoria adecuado y el problema escalado cumple el límite de tiempo impuesto por la ley de Gustafson, se puede incrementar aún más el tamaño del problema, dando una solución mejor o más adecuada.

Con esta filosofía se desarrolló un modelo de memoria limitada. La idea es resolver el problema más grande posible, limitado sólo por la capacidad de memoria disponible. Este modelo puede resultar en un incremento menor en el tiempo de ejecución para obtener performance escalable.

Sea $M$ el requerimiento de memoria de un problema dado, y $W$ la carga de trabajo computacional. Estos factores se encuentran relacionados de varias maneras, dependiendo del espacio de direcciones y las restricciones de arquitectura. Sea $W = g(M)$ o $M = g^{-1}(W)$.

En un multicomputador, la capacidad de memoria total crece linealmente con el número de nodos disponibles. Sea $W = \sum_{i=1}^{q} W_i$ la carga de trabajo para ejecución secuencial en un solo nodo, y $W^* = \sum_{i=1}^{q^*} W_i^*$ la carga de trabajo escalada para ejecución en $p$ nodos, donde $q^*$ es el máximo grado de paralelismo del problema escalado. El requerimiento de memoria para un nodo activo está limitado por $g^{-1}(\sum_{i=1}^{q} W_i)$.

El *speedup de memoria fija* [182] se define de manera similar al de la Ec. 4.7:

$$S_p^* = \frac{\sum_{i=1}^{q^*} W_i^*}{\sum_{i=1}^{q^*} \frac{W_i^*}{i} \lceil \frac{i}{p} \rceil + Q(p)} \tag{4.10}$$

La carga de trabajo para ejecución secuencial en un solo procesador es independiente del tamaño del problema o el tamaño del sistema. Por lo tanto, se puede escribir $W_1 = W_1' = W_1^*$ en los tres modelos. Considerando el caso especial de dos modos operacionales (*secuencial* y *perfectamente paralelo*), la memoria mejorada está relacionada con la carga de trabajo escalada por $W_p^* = g^*(p\,M)$, donde $p\,M$ es la capacidad de memoria incrementada para un multicomputador $p$-nodo. Además, $g^*(p\,M) = G(p)g(M) = G(p)W_p$, donde $W_p = g(M)$ y $g^*$ es una función homogénea. El factor $G(p)$ refleja el incremento en la carga de trabajo cuando la memoria crece $p$ veces. Luego, se puede escribir la Ec. 4.10 con la suposición de que $W_i = 0$ si $i \neq 1$ e $i \neq p$, y $Q(p) = 0$:

$$S_p^* = \frac{W_1^* + W_p^*}{W_1^* + W_p^*/p} = \frac{W_1 + G(p)\,W_p}{W_1 + G(p)W_p/p} \tag{4.11}$$

Hablando rigurosamente, este modelo de speedup es válido bajo dos suposiciones: (1) el conjunto de todas las memorias formen un espacio de direcciones global (en otras palabras, se asume un espacio de memoria distribuida compartida); (2) todas las áreas de memoria disponible son usadas para el problema escalado.

Hay tres casos especiales en que puede aplicarse la Ec. 4.11:

- *Caso 1:* $G(p) = 1$. Corresponde al caso de tamaño del problema fijo. El speedup de memoria fija se vuelve equivalente a la ley de Amdahl, esto es, las Ec. 4.5 y 4.11 son equivalentes en este caso.

- *Caso 2:* $G(p) = p$. Se aplica al caso en que la carga de trabajo crece $p$ veces cuando la memoria crece $p$ veces. Así, la Ec. 4.11 es idéntica a la ley de Gustafson (Ec. 4.8) con un tiempo de ejecución fijo.

- *Caso 3:* $G(p) > p$. Corresponde a la situación donde la carga de trabajo computacional crece más rápido que los requerimientos de memoria. Luego, el modelo de memoria fija (Ec. 4.11) dará un speedup mayor que el de tiempo fijo (Ec. 4.8).

Este análisis indica que las leyes de Amdahl y Gustafson son casos especiales del modelo de memoria fija. Por otra parte, cuando la computación crece más rápido que los requerimientos de memoria (como suele pasar en algunas simulaciones científicas y aplicaciones ingenieriles), el modelo de memoria fija (Figura 4.6) puede dar un speedup mayor (esto es, $S_p^* \geq S_p' \geq S_p$) y mejor utilización de recursos [182].

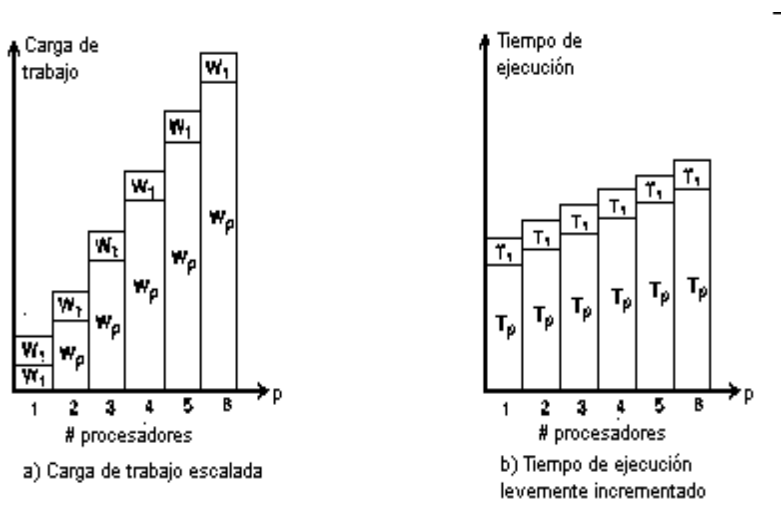


Figura 4.6: Modelo de speedup de speedup escalado usando memoria fija

El modelo de memoria fija también asume una carga de trabajo escalada y permite un ligero incremento en el tiempo de ejecución. El incremento en la carga de trabajo (tamaño del problema) está limitada por la memoria. El crecimiento en el tamaño de máquina está restringido por el incremento de las demandas de comunicación cuando el número de procesadores se vuelve muy grande. El modelo de tiempo fijo puede acercarse mucho al de memoria fija si la memoria disponible es utilizada totalmente.

## 4.12 Escalabilidad de Sistemas Paralelos

Salvo excepciones, el número de procesadores es un límite superior sobre el speedup que puede ser alcanzado por un sistema paralelo. El speedup es 1 para un solo procesador, pero si se usan más, usualmente es menor que el número de procesadores.

**Ejemplo 4.6** *Speedup y eficiencia como funciones del número de procesadores.*

Sea el problema de sumar $n$ números en un hipercubo $p$-procesador. Asumiendo que toma una unidad de tiempo tanto sumar dos números como comunicar un número entre dos procesadores directamente conectados, entonces sumar los $n/p$ números locales a cada procesador toma tiempo $n/p - 1$. Luego, las $p$ sumas parciales toman $\log p$ pasos, cada uno con 1 suma y 1 comunicación. Así, el tiempo de ejecución paralelo total $T_p = n/p - 1 + 2\log p$. Para valores grandes de $n$ y $p$ puede ser aproximado por

$$T_p = \frac{n}{p} + 2\log p \tag{4.12}$$

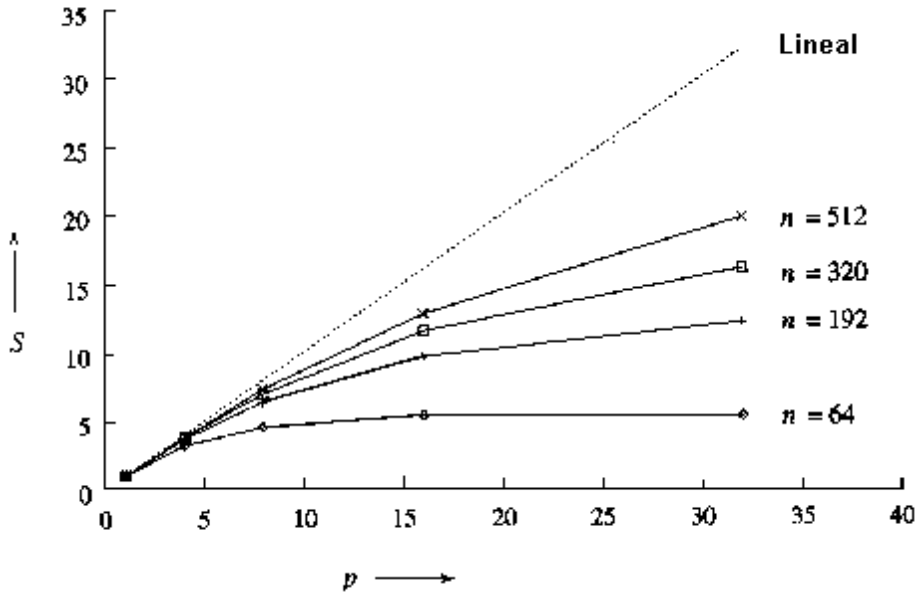


Figura 4.7: Speedup versus número de procesadores para sumar una lista de números en un hipercubo

| $n$ | $p=1$ | $p=4$ | $p=8$ | $p=16$ | $p=32$ |
|---|---|---|---|---|---|
| 64 | 1.0 | .80 | .57 | .33 | .17 |
| 192 | 1.0 | .92 | .80 | .60 | .38 |
| 320 | 1.0 | .95 | .87 | .71 | .50 |
| 512 | 1.0 | .97 | .91 | .80 | .62 |

Tabla 4.1: Eficiencia como función de $n$ y $p$ para sumar $n$ números en un hipercubo $p$-procesador

El tiempo de ejecución secuencial es $n-1$, y puede aproximarse por $n$, por lo que las expresiones para el speedup y la eficiencia son:

$$S = \frac{n\,p}{n + 2p\log p} \tag{4.13}$$

$$E = \frac{n}{n/p + 2p\log p} \tag{4.14}$$

Estas expresiones pueden usarse para calcular el speedup y eficiencia para cualquier par de $n$ y $p$. La Figura 4.7 muestra las curvas de speedup para distintos valores de $n$ y $p$, mientras la Tabla 4.1 muestra las eficiencias correspondientes [209]. □

Este ejemplo muestra dos hechos importantes. Primero, para una instancia de problema dada el speedup no crece linealmente con $p$ sino que tiende a saturarse y la curva se achata (*ley de Amdahl*), haciendo caer la eficiencia. Segundo, una instancia más grande del mismo problema brinda mayor speedup y eficiencia para el mismo número de procesadores, aunque ambos sigan cayendo con $p$ creciente. Estos dos fenómenos, mostrados para un caso particular, son comunes a una gran clase de sistemas paralelos.

Dado que el número creciente de procesadores reduce la eficiencia, y el tamaño creciente de la computación la incrementa, debería ser posible mantener fija la eficiencia creciendo ambos factores simultáneamente. Por ejemplo, en la Tabla 4.1, la eficiencia al sumar 64 números en 4 procesadores es 0.80. Si se pasa a $p = 8$ escalando el tamaño del problema a 192 valores, la eficiencia se mantiene a 0.80, al igual que para $p = 16$ y $n = 512$. Esta capacidad de mantener fija la eficiencia creciendo simultáneamente el número de procesadores y el tamaño del problema es exhibida por varios sistemas paralelos, llamados *sistemas escalables*.

La *escalabilidad* de un sistema paralelo es una medida de su capacidad de incrementar el speedup en proporción al número de procesadores. Refleja la capacidad del sistema para aumentar los recursos de procesamiento efectivamente. El análisis de escalabilidad determina si el procesamiento paralelo de un problema dado puede ofrecer la mejora de performance deseada. Un sistema paralelo puede usarse para resolver problemas arbitrariamente grandes en un tiempo fijo si y solo si se permite que su patrón de carga de trabajo crezca linealmente. A veces, aún si se alcanza un tiempo mínimo con más procesadores, la utilización o eficiencia puede ser muy pobre.

El análisis de escalabilidad puede usarse para elegir la mejor combinación algoritmo-arquitectura para un problema bajo distintas restricciones sobre el crecimiento del tamaño del problema y el número de procesadores. O para predecir la performance de un algoritmo paralelo y una arquitectura para un gran número de procesadores a partir de la performance conocida en menos. Para un tamaño de problema fijo, puede usarse también para determinar el número óptimo de procesadores y el speedup máximo posible que puede obtenerse. También puede predecir el impacto de cambiar la tecnología de hardware sobre la performance y ayudar a diseñar mejores arquitecturas paralelas para resolver varios problemas [211, 264].

Recordar que un sistema paralelo de costo óptimo tiene eficiencia $\Theta(1)$; la escalabilidad y la optimalidad de costo de los sistemas paralelos está relacionada. Un sistema paralelo escalable siempre puede hacerse de costo óptimo si el número de procesadores y el tamaño del problema se eligen apropiadamente. El Ejemplo 4.5 muestra que el sistema paralelo para sumar $n$ números en un hipercubo $p$-procesador es de costo óptimo cuando $n = \Omega(p \log p)$. El siguiente Ejemplo muestra que el mismo sistema es escalable si cuando $p$ crece, $n$ es incrementado en proporción a $\Theta(p \log p)$.

**Ejemplo 4.7** *Escalabilidad de sumar n números en un hipercubo*

Para la suma de costo óptimo de $n$ números en un hipercubo $p$-procesador, $n = \Omega(p \log p)$. Como muestra la Tabla 4.1, la eficiencia es 0.80 para $n = 64$ y $p = 4$. En este punto, la relación entre $n$ y $p$ es $n = 8p \log p$. Si $p = 8$, entonces $8p \log p = 192$. La Tabla 4.1 muestra que la eficiencia es 0.80 con $n = 192$ para $p = 8$. De manera similar, para $p = 16$, la eficiencia es 0.80 para $n = 8p \log p = 512$. Así, este sistema paralelo se mantiene de costo óptimo en una eficiencia de 0.80 si $n$ es incrementado como $8p \log p$ [209]. □

### 4.12.1 Isoeficiencia

Los métodos tradicionales para evaluar algoritmos secuenciales no son adecuados para analizar la performance de combinaciones algoritmo-arquitectura. Se han desarrollado una cantidad de métricas para estudiar la escalabilidad de algoritmos y arquitecturas [146, 149, 336, 339, 366]. En [211], Kumar y Gupta brindan un *survey* de distintos métodos de análisis de escalabilidad. La función de *isoeficiencia* es una de tales métricas, y es útil para evaluar la performance de una gran variedad de combinaciones.

En un sistema paralelo escalable la eficiencia puede mantenerse fija cuando $p$ crece mientras

Para hablar de la métrica de *isoeficiencia*, que permite determinar cuantitativamente el grado de escalabilidad, es importante recordar los conceptos de *tamaño del problema* y *función de overhead*. El tiempo de ejecución paralelo puede ser expresado como una función del tamaño del problema, la función de overhead y el número de procesadores. Reescribiendo la Ec. 4.2, se tiene la siguiente expresión para el tiempo de ejecución paralelo:

$$T_p = \frac{W + T_o(W, p)}{p} \tag{4.15}$$

Luego, el speedup y la eficiencia son:

$$S = \frac{W}{T_p} = \frac{W\,p}{W + T_o(W, p)} \tag{4.16}$$

$$E = \frac{S}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + T_o(W, p)/W} \tag{4.17}$$

En la Ec. 4.17, si $W$ se mantiene constante y $p$ aumenta, $E$ decrece pues el overhead $T_o$ crece con $p$. Si $W$ crece manteniendo $p$ fijo, entonces *para sistemas paralelos escalables* $E$ es mayor. Esto se debe a que $T_o$ crece más lento que $\Theta(W)$ para $p$ fijo. Para tales sistemas, la eficiencia puede mantenerse en un valor deseado (entre 0 y 1) para $p$ creciente, si $W$ también es incrementado.

Para distintos sistemas, $W$ debe crecer a diferentes velocidades con respecto a $p$ para mantener una eficiencia constante. Por ejemplo, en algunos casos, $W$ podría necesitar crecer como una función exponencial de $p$. Tales sistemas paralelos son *pobremente escalables*, ya que es difícil obtener buenos speedups para un número grande de procesadores a menos que el tamaño del problema sea enorme. Por otro lado, si $W$ necesita crecer solo linealmente con respecto a $p$, el sistema es *altamente escalable*, porque pueden obtenerse speedups proporcionales al número de procesadores para tamaños de problema razonables.

Para los sistemas paralelos escalables, $E$ puede ser mantenida en un valor fijo si $T_o/W$ en la Ec. 4.17 se mantiene constante. Para un valor deseado $E$ de eficiencia,

$$E = \frac{1}{1 + T_o(W,p)/W} \quad \Rightarrow \quad \frac{T_o(W,p)}{W} = \frac{1-E}{E}$$

$$W = \frac{E}{1-E} T_o(W,p) \tag{4.18}$$

Sea $K = E/(1-E)$ una constante dependiente de la eficiencia a ser mantenida. Dado que $T_o$ es función de $W$ y $p$, la Ec. 4.18 puede reescribirse como

$$W = K\, T_o(W,p) \tag{4.19}$$

De la Ec. 4.19, el tamaño del problema $W$ puede obtenerse usualmente como función de $p$ por manipulaciones algebraicas. Esta es la *función de isoeficiencia del sistema paralelo* y dicta la velocidad de crecimiento de $W$ necesaria para mantener la eficiencia fija cuando $p$ crece.

La función de isoeficiencia determina la facilidad con la que un sistema paralelo puede mantener una eficiencia constante y por lo tanto obtener speedups crecientes en proporción al número de procesadores. Una función de isoeficiencia chica significa que pequeños incrementos en $W$ son suficientes para la utilización eficiente de un número creciente de $p$, indicando que el sistema es altamente escalable. Sin embargo, una función de isoeficiencia grande indica un sistema paralelo pobremente escalable. La función de isoeficiencia no existe para sistemas paralelos no escalables, porque en ellos la eficiencia no puede ser mantenida en ningún valor constante cuando $p$ crece, sin importar cuan rápido se incremente $W$ [140].

**Ejemplo 4.8** *Función de isoeficiencia de la suma de números en un hipercubo.*

Sea el problema de sumar $n$ números en un hipercubo $p$-procesador. Bajo las suposiciones del Ejemplo 4.6, el tiempo de ejecución paralela es aproximadamente $n/p + 2\log p$. La

misma tarea puede realizarse secuencialmente en tiempo aproximadamente $n$. Así, del tiempo $n/p + 2\log p$ que cada procesador usa en la ejecución paralela, aproximadamente $n/p$ se usa para trabajo útil. El tiempo $(2\log p)$ restante por procesador lleva a un overhead total de $T_o \approx p\,(\frac{n}{p} + 2\log p) - n = 2p\log p$. Sustituyendo $T_o$ por $2p\log p$ en la Ec. 4.19 se tiene

$$W = 2K\,p\log p \tag{4.20}$$

Luego, la función de isoeficiencia asintótica para este sistema paralelo es $\Theta(p\log p)$. Esto significa que si el número de procesadores es incrementado de $p$ a $p'$, el tamaño del problema (en este caso $n$) debe ser aumentado por un factor de $(p'\log p')/(p\log p)$ para tener la misma eficiencia que en $p$ procesadores. En otras palabras, incrementar el número de procesadores por un factor de $p'/p$ requiere que $n$ crezca por un factor de $(p'\log p')/(p\log p)$ para que el speedup crezca por un factor de $p'/p$ [209]. □

En este caso, el *overhead de comunicación* es función sólo de $p$. En general, puede depender tanto del tamaño del problema como del número de procesadores. Una función de overhead típica puede tener varios términos diferentes de distintos órdenes de magnitud con respecto a $p$ y $W$. En tal caso, puede ser difícil (o imposible) obtener la función de isoeficiencia como una función cerrada de $p$. Por ejemplo, sea un sistema paralelo hipotético para el cual $T_o = p^{3/2} + p^{3/4}\,W^{3/4}$; luego, la Ec. 4.19 puede reescribirse como $W = K\,p^{3/2} + K\,p^{3/4}\,W^{3/4}$. Es difícil resolver esta ecuación para $W$ en términos de $p$.

Dado que la condición para eficiencia constante es que $T_o/W$ se mantenga fijo, cuando $p$ y $W$ crecen la eficiencia es no decreciente cuando ninguno de los términos de $T_o$ crece más rápido que $W$. Si $T_o$ tiene múltiples términos, se balancea $W$ contra cada uno de ellos y se computan las funciones de isoeficiencia respectivas para los términos individuales. La componente de $T_o$ que requiere que el tamaño del problema crezca a la mayor velocidad con respecto a $p$ determina la función de isoeficiencia asintótica general del sistema paralelo. El siguiente ejemplo ilustra la técnica del análisis de isoeficiencia.

**Ejemplo 4.9** *Función de isoeficiencia de un sistema paralelo con una función de overhead compleja.*

Sea un sistema paralelo para el cual $T_o = p^{3/2} + p^{3/4}\,W^{3/4}$. Usando sólo el primer término de $T_o$ en la Ec. 4.19 se tiene

$$W = K\,p^{3/2} \tag{4.21}$$

Usando sólo el segundo término se tiene la siguiente relación entre $W$ y $p$:

$$W = K\,p^{3/2} + K\,p^{3/4}\,W^{3/4} \quad \Rightarrow \quad W^{1/4} = K\,p^{3/4}$$

$$W = K^4 p^3 \tag{4.22}$$

Para asegurar que la eficiencia no cae cuando $p$ crece, el primero y segundo término de la función de overhead requieren que $W$ crezca como $\Theta(p^{3/2})$ y $\Theta(p^3)$ respectivamente. La más grande asintóticamente de las dos velocidades, $\Theta(p^3)$, da la función de isoeficiencia asintótica general de este sistema paralelo, dado que subsume la velocidad dictada por el otro término [209]. □

En una expresión simple, la función de isoeficiencia captura las características de un algoritmo paralelo así como de la arquitectura en la que se lo implementa. Después de realizar el análisis de isoeficiencia, se puede testear la performance de un programa paralelo sobre unos pocos procesadores y predecir su comportamiento sobre un número mayor. Además, puede caracterizar la cantidad de paralelismo inherente en un algoritmo paralelo, y puede usarse para estudiar el comportamiento de un sistema paralelo con respecto a cambios en parámetros de hardware tales como la velocidad de los procesadores y canales de comunicación.

En las siguientes subsecciones se establecen relaciones entre diversos factores y la función de isoeficiencia. Este tema se encuentra ampliamente cubierto en [209, 140].

### Optimalidad de costo y la función de isoeficiencia

Un sistema paralelo es de costo óptimo si y sólo si el producto del número de procesadores y el tiempo de ejecución paralelo es proporcional al tiempo de ejecución del algoritmo secuencial más rápido sobre un solo procesador. Esto es:

$$p\,T_p = \Theta(W) \tag{4.23}$$

Sustituyendo $T_p$ en la Ec. 4.15 se tiene

$$W + T_o(W, p) = \Theta(W)$$

$$T_o(W, p) = O(W) \tag{4.24}$$

$$W = \Omega(T_o(W, p)) \tag{4.25}$$

Esto sugiere que un sistema paralelo es de costo óptimo si y solo si su función de overhead no excede asintóticamente el tamaño del problema. Esto es similar a la condición

para mantener una eficiencia fija al crecer el número de procesadores. Si la Ec. 4.19 da una función de isoeficiencia $f(p)$, entonces de la Ec. 4.25 se tiene que debe satisfacerse la relación $W = \Omega(f(p))$ para asegurar la optimalidad de costo al escalar el sistema paralelo.

### Una cota inferior para la función de isoeficiencia

Una función de isoeficiencia más chica indica mayor escalabilidad. Luego, un sistema paralelo idealmente escalable debe tener la función de isoeficiencia más baja posible. Si un problema consiste de $W$ unidades de trabajo, no pueden usarse más de $W$ procesadores con optimalidad de costo. Si el tamaño del problema crece a una velocidad menor que $\Theta(p)$ cuando el número de procesadores se incrementa, entonces $p$ eventualmente excederá $W$. Aún para un sistema paralelo ideal sin comunicación u otro overhead, la eficiencia caerá porque los procesadores agregados más allá de $p = W$ estarán ociosos. Asintóticamente, el tamaño del problema debe crecer al menos tan rápido como $\Theta(p)$ para mantener fija la eficiencia; por lo tanto, $\Omega(p)$ es la cota inferior asintótica para la función de isoeficiencia. Luego, la función de isoeficiencia de un sistema paralelo idealmente escalable es $\Theta(p)$.

### El grado de concurrencia y la función de isoeficiencia

La cota inferior de $\Omega(p)$ es impuesta a la función de isoeficiencia de un sistema paralelo por el número de operaciones que pueden realizarse concurrentemente. El grado de concurrencia ($C(W)$) es una medida del número de operaciones que un algoritmo puede realizar en paralelo en un problema de tamaño $W$ y es independiente de la arquitectura. Para un problema de tamaño $W$, a lo sumo $C(W)$ procesadores pueden emplearse de manera efectiva.

Por ejemplo, usando eliminación Gaussiana para resolver un sistema de $n$ ecuaciones con $n$ variables, la cantidad total de computación es $\Theta(n^3)$. Pero, las $n$ variables deben ser eliminadas una después de la otra, y elminar una variable requiere $\Theta(n^2)$ computaciones. Así, a lo sumo $\Theta(n^2)$ procesadores pueden mantenerse ocupados en cualquier momento. Como $W = \Theta(n^3)$ para este problema, $C(W)$ es $\Theta(W^{2/3})$, y a lo sumo $\Theta(W^{2/3})$ procesadores pueden usarse eficientemente. Por otra parte, dados $p$ procesadores, el tamaño del problema debería ser al menos $\Omega(p^{3/2})$ para usarlos todos. Por lo tanto, la función de isoeficiencia de esta computación debido a la concurrencia es $\Theta(p^{3/2})$.

La función de isoeficiencia debido a la concurrencia es óptima (esto es, $\Theta(p)$) solo si el grado de concurrencia del algoritmo paralelo es $\Theta(W)$. Si el grado de concurrencia de un algoritmo es menor que $\Theta(W)$, entonces la función de isoeficiencia debido a la concurrencia es peor (esto es, más grande) que $\Theta(p)$. En tales casos, la función de isoeficiencia general de un sistema paralelo está dado por el máximo de las funciones de isoeficiencia debido a la concurrencia, comunicación y otros overheads.

### Tiempo de ejecución mínimo y Tiempo de ejecución mínimo de costo óptimo

Con frecuencia interesa saber cuál es el mínimo tiempo posible de ejecución de un algoritmo, dado que el número de procesadores no es una restricción. A medida que crece $p$ para un problema dado, o el tiempo de ejecución paralelo sigue decreciendo aproximándose asintóticamente a un mínimo, o comienza a subir después de llegar al mínimo. Se puede determinar el tiempo paralelo mínimo $T_p^{\min}$ para un $W$ dado diferenciando $T_p$ respecto a $p$ e igualando la derivada a 0 (asumiendo una función diferenciable con respecto a $p$). El número de procesadores para el cual $T_p$ es mínimo está determinado por

$$\frac{d}{d\,p}T_p = 0 \tag{4.26}$$

Si $p_0$ es el valor que satisface esta ecuación, el valor de $T_p^{\min}$ puede determinarse sustituyendo $p_0$ por $p$ en la expresión de $T_p$.

**Ejemplo 4.10** *Mínimo tiempo de ejecución para sumar n números en un hipercubo*

Puede verse que $T_p^{\min} = 2 \log n$ [209]. □

En este ejemplo, el producto procesador-tiempo para $p = p_0$ es $\Theta(n \log n)$, que es mayor que la complejidad serial $\Theta(n)$. Por lo tanto, el sistema paralelo no es de costo óptimo para el valor de $p$ que da el tiempo de ejecución paralelo mínimo.

Sea $T_p^{costo\_opt}$ el tiempo mínimo en el cual un problema puede ser resuelto por un sistema paralelo de costo óptimo. De la equivalencia de la optimalidad de costo y la función de isoeficiencia, si la función de isoeficiencia de un sistema paralelo es $\Theta(f(p))$, entonces un problema de tamaño $W$ puede ser resuelto con costo óptimo si y solo si $W = \Omega(f(p))$.

En otras palabras, dado un problema de tamaño $W$, una solución de costo óptimo requiere que $p = O(f^{-1}(W))$. Como el tiempo de ejecución paralelo es $\Theta(W/p)$ para un sistema paralelo de costo óptimo (Ec. 4.23), la cota inferior sobre el runtime paralelo para resolver un problema de tamaño $W$ con costo óptimo es

$$T_p^{costo\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right) \tag{4.27}$$

**Ejemplo 4.11** *Tiempo de ejecución mínimo de costo óptimo para sumar n números en un hipercubo*

Según el Ejemplo 4.8, la función de isoeficiencia $f(p)$ de este sistema es $\Theta(p \log p)$. Si $W = n = f(p) = p \log p$, entonces $\log n = \log p - \log\log p$. Ignorando el doble logaritmo, $\log n \approx \log p$. Si $n = f(p) = p \log p$, entonces $p = f^{-1}(n) = n/\log p \approx n \log n$. Por lo tanto, $f^{-1}(W) = \Theta(n/\log n)$. Como consecuencia de la relación entre optimalidad de costo y la función de isoeficiencia, el número máximo de procesadores que pueden ser usados con costo óptimo es $\Theta(n/\log n)$. Usando $p = n/\log n$ en la Ec. 4.12 se tiene [209]

$$T_p^{costo\_opt} = \log n + 2 \log\left(\frac{n}{\log n}\right) = 3 \log n - 2 \log\log n \tag{4.28}$$

□

En este caso tanto $T_p^{\min}$ como $T_p^{costo\_opt}$ son $\Theta(\log n)$, y así una solución de costo óptimo es también la más rápida asintóticamente. El tiempo de ejecución paralelo no puede ser reducido asintóticamente usando un valor de $p$ mayor que el sugerido por la función de isoeficiencia para un tamaño de problema dado (debido a la equivalencia entre optimalidad de costo y la función de isoeficiencia). Esto no es verdad para sistemas paralelos en general, y normalmente $T_p^{costo\_opt} > \Theta(T_p^{\min})$. Los sistemas paralelos para los cuales el tiempo de ejecución puede ser reducido por un orden de magnitud usando un número de procesadores asintóticamente más grande que el indicado por la función de isoeficiencia son raros.

Al calcular el mínimo tiempo de ejecución para un sistema paralelo, es importante recordar que el máximo número de procesadores que pueden utilizarse está limitado por $C(W)$. Es bastante posible que $p_0$ sea mayor que $C(W)$ para un sistema paralelo; en tales casos, el valor de $p_0$ no tiene significado, y $T_p^{\min}$ está dado por $\frac{W+T_o(W,C(W))}{C(W)}$.

### 4.12.2 Isospeed

Otra métrica utilizada en el estudio de la escalabilidad de los sistemas paralelos es *isospeed*, definida formalmente en [339, 341, 342], y estudiada en [232, 338].

Una computación paralela puede especificarse por una combinación algoritmo-máquina $C = (A, M)$, es decir, un algoritmo $A$ implementado sobre un sistema de cómputo paralelo $M$. Dos objetivos del paralelismo son obtener menores tiempos de ejecución y resolver problemas más grandes. En particular, el análisis de escalabilidad se basa en el tiempo de ejecución paralelo y el tamaño del problema. El *tamaño del problema* ($W$), cantidad de operaciones básicas realizadas por un algoritmo $A$ en una máquina $M$, da una medida del trabajo útil llevado a cabo para resolver un problema; también suele llamarse *trabajo* de una computación paralela $C$. El *tiempo de ejecución* de $A$ cuando el tamaño del problema es $W$ y hay $p$ procesadores en $M$ es $T_p(W, p)$. El parámetro $p$ es el *tamaño* de la máquina paralela $M$. Para combinar las consideraciones sobre el tamaño del problema y el tiempo de ejecución, se define la *velocidad* de $C$ como el cociente entre $W$ y $T_p(W, p)$,

$$S(W,p) = \frac{W}{T(W,p)}$$

La *velocidad promedio* es la velocidad dividida por el tamaño de la máquina:

$$\overline{S}(W,p) = \frac{T(W,p)}{p} = \frac{W}{T(W,p)\,p}$$

La velocidad representa una cantidad que idealmente se incrementaría con el número de procesadores, mientras la velocidad promedio es una medida de eficiencia del sistema de cómputo subyacente.

El tiempo de ejecución paralelo puede dividirse en dos partes: tiempo de cómputo y overhead del procesamiento paralelo. Esto es, se puede escribir

$$T(W,p) = \frac{W + T_o(W,p)}{p}$$

donde $T_o(W,p)$ es el overhead de implementar $A$ en $M$, por lo que

$$\overline{S}(W,p) = \frac{W}{W + T_o(W,p)}$$

Usualmente, si $p$ crece mientras $W$ se mantiene constante, la velocidad promedio decrece debido al incremento de la función de overhead con $p$. Por otro lado, la velocidad promedio crece con $W$ si $p$ es fijo, dado que incrementar el tamaño del problema usualmente reduce el cociente overhead/computación. De esta forma, es posible mantener constante la velocidad promedio (o mantener la velocidad linelamente proporcional a $p$) si se permite que $W$ crezca con $p$; esto es lo que significa cómputo paralelo escalable.

La *escalabilidad isospeed* de una combinación algoritmo máquina $C = (A, M)$ cuando el tamaño de máquina es escalado de $p$ a $p'$ y el tamaño de problema se permite que crezca de $W$ a $W'$ se define como

$$\psi(p,p') = \left(\frac{W}{p}\right) \Big/ \left(\frac{W'}{p'}\right) = \frac{p'\,W}{P\,W'} \tag{4.29}$$

Este cociente mide esencialmente cómo debería incrementarse la cantidad de trabajo por procesador cuando el tamaño de máquina es escalado de $p$ a $p'$ para mantener la misma velocidad promedio. La forma en que $W$ crece con $p$ para mantener constante

$\overline{S}(W,p)$ está determinada por la naturaleza de la combinación algoritmo-máquina y es el centro del análisis de escalabilidad. Esencialmente se debe encontrar $W = f(p)$ tal que

$$\overline{S}(W,p) = \frac{f(p)}{T_p(f(p),p)\,p} = \Theta(1)$$

o al menos encontrar la velocidad de crecimiento de $W = f(p)$ requerida para mantener una velocidad promedio constante cuando $p$ crece. Una velocidad de crecimiento chica de $f(p)$ implica alta escalabilidad, mientras una velocidad grande significa escalabilidad pobre. La función $f(p)$ tiene la siguiente implicación: si $W = w(f(p))$ ($W = o(f(p))$, respectivamente), $\overline{S}(W,p)$ es una función creciente (decreciente, respectivamente) de $p$ cuando $p \to \infty$. Además, si se elige $W = f(p)$ se tiene $\overline{S}(W,p) = \overline{S}(W',p')$, esto es

$$\frac{W}{T_p(W,p)\,p} = \frac{W'}{T_p(W',p')\,p'} \Longrightarrow \psi(p,p') = \frac{T_p(W,p)}{T_p(W',p')}$$

Esto indica que la escalabilidad isospeed brinda una manera de medir la degradación de performance de computaciones paralelas más grandes (es decir, mayores tamaños de problema y más procesadores) versus computaciones más chicas (tamaños de problema menores y menos procesadores). Un valor más grande (más chico, respectivamente) de $\psi(p,p')$ implica menor (mayor, respectivamente) degradación de performance.

En [232] se analiza la escalabilidad isospeed presentando un modelo probabilístico donde se modelizan los algoritmos paralelos sobre multiprocesadores usando grafos de precedencia, y se tratan los tiempos de ejecución como variables aleatorias. Se derivan las velocidades de crecimiento del número de tareas paralelas para clases típicas de grafos de tarea (computaciones iterativas, árboles de búsqueda, algoritmos de particionado, etc), así como la escalabilidad isospeed para mantener la velocidad promedio constante. Además de los resultados analíticos, se presentan numerosos datos numéricos. Un concepto importante es que mientras una computación paralela puede hacerse escalable incrementando el tamaño del problema junto con el tamaño del sistema, es en realidad la *cantidad de paralelismo* la que debería escalar con el tamaño del sistema.

### 4.12.3 Relación entre escalabilidad y tiempo de ejecución

La predicción de escalabilidad y la relación entre escalabilidad y tiempo de ejecución han sido estudiadas en [338, 332, 334, 333]. Los resultados teóricos y experimentales muestran que la escalabilidad combinada con el tiempo de ejecución inicial pueden brindar buena predicción de performance, en términos de tiempos de ejecución.

Integrar la escalabilidad en la predicción de performance requiere un entendimiento profundo de su relación con el tiempo de ejecución. Sin embargo, tradicionalmente fueron

estudiados por separado. En [334], Sun introdujo el concepto de *comparación de rango*. A diferencia de la comparación de tiempo de ejecución convencional en la cual la performance es comparada en una plataforma paralela dada y un sistema y tamaño de problema especificado, la comparación de rango testea la performance en un amplio rango de combinaciones de ensambles y tamaños de problema a través de análisis de *crossing points*.

La idea es sencilla: encontrar el primer crossing point de performance superior/inferior. Antes de llegar a ese punto, en una gran cantidad de tamaños de sistema y problema, un programa rápido se mantendrá rápido y uno lento se mantendrá lento. El artículo citado brinda una forma de encontrar el crossing point a través del análisis de escalabilidad.

Un resultado teórico importante es que *si una combinación algoritmo-máquina es más rápido en el estado inicial y tiene una mejor escalabilidad que otras combinaciones, entonces se mantendrá superior en el rango escalable.*

La comparación de rango representa un desafío mayor cuando la combinación inicial más rápida tiene una menor escalabilidad. Cuando el tamaño del sistema se escala, un código originalmente más rápido pero menos escalable puede volverse más lento que uno con mejor escalabilidad. Encontrar el *punto de cruce* es crítico para obtener performance óptima. Una definición informal basada en isospeed es la siguiente:

Sea que la combinación algoritmo-máquina 1 tiene tiempo de ejecución $t$, escalabilidad $\Phi(p, p')$ y tamaño de problema escalado $W'$. Y sea que la combinación algoritmo-máquina 2 tiene tiempo de ejecución $T$, escalabilidad $\Psi(p, p')$ y tamaño de problema escalado $W^*$. Si $t_p(W) = \alpha\, T_p(W)$ en el tamaño de ensamble inicial $p$ y el tamaño de problema $W$ para algún $\alpha > 1$, entonces $p'$ es un crossing point de las combinaciones 1 y 2 si y solo si

$$\frac{\Phi(p, p')}{\Psi(p, p')} > \alpha \tag{4.30}$$

De hecho, cuando $\Phi(p, p') > \alpha\Psi(p, p')$, se tiene $t_{p'}(W') < T_{p'}(W^*)$. Dado que $\alpha > 1$ la combinación 2 tiene un tiempo de ejecución menor en el estado inicial $t_p(W) > T_p(W)$. Este cambio rápido/lento en el tiempo de ejecución da el significado de crossing point.

Otro resultado relevante es el siguiente: *si se asume que la combinación algoritmo-máquina 1 tiene un tiempo de ejecución mayor que la combinación 2 en el estado inicial, entonces el tamaño de ensamble escalado $p'$ no es un crossing point escalado si y solo si la combinación 1 tiene un tiempo mayor que el de la combinación 2 para resolver cualquier problema escalado $W^\dagger$ tal que $W^\dagger$ está entre $W'$ y $W^*$ en $p'$, donde $W'$ y $W^*$ son el tamaño de problema escalado de la combinación 1 y 2 respectivamente.*

Esto da la condición necesaria para la comparación de rango de la computación escalable: $p'$ no es un crossing point de $p$ si y solo si la relación rápido/lento de los códigos no cambia para ningún tamaño de problema escalado dentro del rango escalable de las dos

combinaciones. Basado en este resultado teórico, con la comparación de escalabilidad se puede predecir la performance relativa de algoritmos en un rango de tamaños de problema y máquina. Esta propiedad de la comparación de escalabilidad es valiosa en la práctica.

### 4.12.4 Otras relaciones e *iso*-métricas

En [246] se presenta un método práctico para estudiar la escalabilidad de combinaciones algoritmo-arquitectura. El mismo trata en forma separada el tiempo de ejecución, la eficiencia y el uso de memoria en un modelo de escalado de precisión crítica, donde el tamaño del problema crece con el número de procesadores, lo que es lo relevante en muchas situaciones. El artículo define medidas cuantitativas y cualitativas de escalabilidad y deriva importantes relaciones entre el tiempo de ejecución y la eficiencia. Por ejemplo, los resultados muestran que la mejor manera de escalar el sistema (para deteriorar lo menos posible sus propiedades) es mantener constante el tiempo de ejecución.

En muchos casos, el tamaño del problema debe incrementarse de manera sublineal con el número de procesadores para mantener constante el tiempo de ejecución, y tiene que crecer en forma superlineal para mantener la eficiencia constante. De hecho, las funciones de isoeficiencia e *isotiempo* son polinomiales. El análisis presentado en [246] utiliza métricas de escalabilidad *isoparamétricas*, esto es, las funciones de isoeficiencia, isotiempo e isomemoria.

Estas funciones determinan la facilidad con la cual el sistema puede mantener los parámetros paralelos sin cambio en proporción al número de procesadores. Usando el modelo de precisión crítica, interesa obtener la solución más precisa sin exceder los límites de memoria. Eso es, el tamaño del problema debería crecer linealmente con respecto al número de procesadores (función de isomemoria). Luego, si interesa preservar la eficiencia, cuanto más cercana esté la función de isoeficiencia a la de isomemoria, más escalable es el sistema con respecto a la eficiencia, pues significa un incremento de tiempo de ejecución y uso de memoria más lento con el número de procesadores. De la misma manera, cuanto más cercana esté la función de isotiempo de la de isomemoria, más escalable es el sistema con respecto al tiempo de ejecución porque esto significa un decrecimiento más lento de la eficiencia y el uso de memoria con el número de procesadores.

Esto brinda una medida cualitativa de la escalabilidad, representando las tres funciones isoparamétricas en un grafo. Más aún, se sabe que un parámetro crece o decrece con una escala dada por una función de crecimiento del tamaño del problema si su correspondiente función isoparamétrica está por debajo o encima de esa función. Un sistema paralelo es perfectamente escalable si es perfectamente escalable con respecto a todos sus parámetros paralelos; todas las funciones isoparamétricas creen linealmente y, por lo tanto, coinciden con la función de isomemoria. Si el sistema paralelo es escalado con el modelo de isomemoria, el tiempo de ejecución y la eficiencia se mantienen constantes.

## 4.13 Análisis de rendimiento. Predicción y *tuning*

El principal objetivo del procesamiento paralelo y distribuido es el rendimiento: un usuario o programador que se plantea la resolución de un problema mediante una aplicación paralela espera obtener mejoras en el rendimiento respecto de una solución secuencial.

Una aproximación es utilizar métodos que permitan la *predicción* de performance del sistema paralelo. En este sentido se han presentado en el Capítulo 1 algunos modelos de computación paralela existente, uno de cuyos objetivos es precisamente la predicción. El tema es abordado en forma extensa en la literatura, especialmente en lo que se refiere a ambientes y herramientas. Pueden citarse a modo de ejemplo [317, 81, 334, 273, 201, 164, 207, 176, 251, 235]

El modelo clásico para el análisis de aplicaciones paralelas se basa en el enfoque "medir y modificar" (*measure and modify*). Para esto se han desarrollado una serie de herramientas de monitorización y visualización, que permiten realizar el *tuning* o sintonización de performance. Entre las herramientas existentes se encuentran *PAT* (Performance Analysis Tool, desarrollada por Cray para el sistema T3E), *TapePVM* (para aplicaciones PVM), *Vampir* (Visualization and Analysis of MPI pRograms, herramienta comercial de Pallas para aplicaciones MPI), *AIMS* (Automated Instrumentation and Monitoring System, conjunto de herramientas para medida y análisis de rendimiento). Generalmente las herramientas basadas en visualización ofrecen información de muy bajo nivel, y es difícil para el usuario comprender todas las relaciones existentes entre los grafos mostrados y relacionarlos con el código fuente. Esto implica la necesidad de un gran nivel de experiencia para poder sacar conclusiones acerca de los problemas [55].

Las técnicas más modernas se refieren al *análisis automático* [256]. En este caso, los objetivos son determinar los problemas de rendimiento y sus causas, relacionar el problema con el código fuente y modificar el código fuente para mejorar el rendimiento de la aplicación, todo esto como un proceso automatizado. Ejemplos se encuentran en Kappa-Pi (Knowledge-based Automatic Parallel Program Analyser for Performance Improvement, desarrollado en la Universidad Autónoma de Barcelona) y Paradyn (de la University of Wisconsin at Madison).

# Parte II

# Balance de Carga y Sorting

# Capítulo 5

# Asignación de tareas y balance de carga

## 5.1 Introducción

Una aplicación paralela define un conjunto de componentes intercomunicados que deben ser alocados en los recursos físicos de la arquitectura de destino. Esto es, la programación paralela agrega una nueva dimensión a la tradicional, ya que no sólo hay que saber *cuándo* se ejecutará una operación, sino que además debe definirse *dónde*, en qué procesador.

La última etapa del diseño de algoritmos paralelos presentada en la Sección 3.6 es el mapeo. Un objetivo de la política de mapeo o asignación de tareas es el *balance de carga* ("*load balancing*" o *LB*) [73]. Este es uno de los aspectos centrales del cómputo paralelo, ya que impacta directamente sobre el uso eficiente de los recursos y las mejoras de performance que se pueden lograr [210]. Desafortunadamente, ningún método ni algoritmo es apropiado para *todas* las aplicaciones.

El problema puede compararse con los encontrados en procesos de distribución de trabajo como el de *scheduling* de actividades para construir un puente, o el flujo de trabajos en sistemas de producción. Para esto deben considerarse varios objetivos:

1. Completar todo el trabajo en el menor tiempo posible.
2. Los trabajadores (*workers*) implican costo, por lo que deben mantenerse ocupados. Esto puede ser simple en períodos con mucho trabajo, pero la distribución debe planearse con cuidado cuando las actividades son escasas.
3. El trabajo debería distribuirse equitativamente entre los *workers*.
4. Se deben respetar las restricciones de precedencia entre las tareas.

El uso desigual de los PEs puede causar una eficiencia pobre o incluso hacer que el tiempo de resolución paralelo sea mayor que el secuencial. Cuando un problema es resuelto ejecutando un algoritmo sobre una máquina paralela, el trabajo debe ser particionado entre los procesadores de manera tal que los recursos del sistema sean utilizados eficientemente y, en algunos casos, minimizando una cierta función de costo [222].

El objetivo abstracto del balance de carga puede definirse como sigue: *Dado un conjunto de tareas que comprenden una computación y un conjunto de computadoras sobre las cuales pueden ejecutarse dichas tareas, encontrar el mapeo de tareas a computadoras que resulte en que cada una tenga una cantidad de trabajo aproximadamente igual.* Un mapeo que balancea la carga de trabajo de los procesadores *típicamente* incrementa la eficiencia global de la computación y reduce su tiempo de ejecución.

El problema de asignación es *NP*-completo para un sistema general con $n$ procesadores, y por lo tanto la tarea de encontrar una asignación de costo mínimo es computacionalmente intratable salvo para sistemas muy chicos. Por esta razón pueden utilizarse enfoques alternativos como la *relajación* (se relajan algunos de los requerimientos o se restringe el problema), el desarrollo de soluciones para casos particulares, la *optimización enumerativa* (uso de métodos enumerativos como programación dinámica y *branch-and-bound*), o la *optimización aproximada* (la utilización de heurísticas que brindan soluciones subóptimas aunque aceptables) [192, 254, 324, 220, 29, 30, 102, 128, 127, 162, 161].

Los problemas de balance de carga difieren principalmente en:

- *Costo de las tareas.* ¿Todas las tareas tienen igual costo? Si no es así, ¿cuándo se conocen los costos (antes de empezar, al crear las tareas, o sólo cuando éstas terminan)? El caso más sencillo se da cuando las tareas tienen todas costo unitario (*branch-free loops*). Es más complicado cuando tienen tiempos diferentes aunque conocidos (multiplicación matriz-vector *sparse*). El peor caso se da cuando los costos no se conocen hasta que finaliza la ejecución (GCM, circuitos).

- *Dependencia de las tareas.* ¿Todas las tareas pueden ejecutar en cualquier orden (incluyendo las paralelas)? Sino, ¿cuándo se conocen las dependencias (antes de empezar, cuando se crean las tareas, o sólo cuando terminan)? Lo más sencillo es que puedan ejecutar en cualquier orden (*dependence free loops*). La complejidad es mayor cuando las tareas tienen alguna estructura predecible (computaciones matriciales), y aumenta aún más cuando la estructura cambia dinámicamente, lenta o rápidamente (búsquedas, LU *sparse*).

- *Localidad.* ¿Es importante para algunas tareas ser planificadas en el mismo procesador (o cerca) para reducir costos de comunicación? ¿Cuándo se conoce la información sobre comunicación entre tareas? El caso más sencillo se tiene cuando las tareas no se comunican (*embarrassingly parallel*). Es más difícil si se comunican con

un patrón predecible (resolución de ecuaciones diferenciales parciales), y aún más cuando el patrón es impredecible (simulación de eventos discretos).

Al considerar el problema de balance de la carga es importante distinguir entre la *descomposición del problema* y el *mapeo de tareas*. La primera involucra la explotación de la concurrencia en el control y el acceso a los datos de un algoritmo. Su resultado es un conjunto de tareas comunicantes que resuelven el problema en paralelo. Estas tareas luego pueden ser mapeadas a las computadoras en la forma que mejor se ajuste al problema.

En [328], Stone sugirió un algoritmo óptimo que resulta eficiente para el problema de asignar tareas a dos procesadores (*two processor problem*) haciendo uso del conocido algoritmo de flujo en redes en grafos de dos terminales. Allí mostró cómo el modelo de flujo en red puede extenderse a sistemas de 3 o más procesadores. A partir de allí, el problema de asignar tareas de una aplicación a los procesadores de un sistema paralelo fue estudiado por muchos investigadores, desarrollándose diferentes técnicas y algoritmos como en [330, 329, 46, 21, 47, 98, 347, 48, 239, 130, 260, 221].

Si bien el balance de carga indica la cantidad de tareas que un procesador ejecutará, no debe olvidarse que la migración de procesos no es un tema menor (por ejemplo, ¿qué sucede con los archivos abiertos por un proceso paralelo a ser migrado?, ¿cuál es el patrón de comunicaciones?, ¿que podría influir a la hora de migrar procesos?).

Además, es necesario tener en cuenta que *en muchos casos (especialmente en problemas irregulares o cuyo tiempo de ejecución es altamente dependiente de las características de los datos) puede no alcanzar con una distribución equitativa del número de tareas.*

En algunos casos el tiempo de cómputo asociado con una tarea dada puede determinarse *a priori*. En tales circunstancias, se puede realizar el mapeo de tareas antes de comenzar la computación (balance de carga *estático*). Para una clase importante y creciente de aplicaciones, la carga de trabajo para una tarea particular puede modificarse en el curso de una computación y no puede estimarse de antemano, y el mapeo debe cambiar *dinámicamente* durante el cómputo [59, 355, 371].

Gran parte de los trabajos sobre balance de carga estático y dinámico conocidos intentan mejorar algoritmos de balance ya existentes y ampliamente utilizados, o están dirigidos a un número reducido de aplicaciones. En particular, muchos de los algoritmos de balance de carga estáticos son diseñados teniendo en cuenta el problema a balancear y la arquitectura sobre la que se ejecutarán. Esta clase de algoritmos necesita un período de "puesta a punto", el cual consume tiempo y debe ser reevaluado a la hora de escalar el tamaño del problema o de modificar alguna de las características de la plataforma de destino. Esto último es muy importante, pues los clusters de PCs o de WorkStations son cada vez más comunes y podrían estar sujetos a constantes actualizaciones derivando en clusters heterogéneos [44].

## 5.2 Grafos de tareas. Scheduling

Un *scheduling* eficiente de un programa paralelo sobre los procesadores es vital para lograr una alta performance [215]. Cuando la estructura del programa en términos de sus tiempos de ejecución de tareas, dependencias, comunicación y sincronización es conocida *a priori*, el scheduling puede realizarse estáticamente en tiempo de compilación. El objetivo es minimizar la longitud del schedule.

Un programa paralelo puede representarse mediante un grafo dirigido sin ciclos (*directed acyclic graph*, DAG) $G = (V, E)$, donde $V$ es el conjunto de nodos ($|V| = v$) y $E$ es el conjunto de arcos ($|E| = e$). Un nodo representa una tarea, y a cada nodo se asocia su *costo de cómputo* ($w(n_i)$) que indica el tiempo de ejecución. Los arcos corresponden a los mensajes de comunicación y restricciones de precedencia entre los nodos. Cada arco tiene asociado un número que indica el tiempo requerido para comunicar los datos de un nodo a otro (*costo de comunicación*, $c_{ij}$). Los nodos fuente y destino de un arco se denominan *padre* e *hijo* respectivamente. En un grafo de tareas, un nodo sin padre se denomina *entry* y uno sin hijo *exit*. Un nodo no puede comenzar su ejecución antes de recibir todos los mensajes de sus padres.

La asignación de tareas a procesadores puede representarse gráficamente mediante un diagrama de Gantt, que indica el tiempo que cada tarea utiliza en ejecución y sobre qué procesador. Esto permite observar la utilización de los procesadores (el porcentaje de tiempo en que ejecutan tareas).

El objetivo del scheduling estático (que puede aplicarse a *problemas estáticos*, con estructura predecible) es asignar los nodos a los procesadores minimizando el schedule y sin violar las restricciones de precedencia. Un schedule es *eficiente* si su longitud es corta y el número de procesadores usados es razonable. Hay varios enfoques que pueden emplearse, incluyendo teoría de colas, teoría de grafos y búsqueda en espacio de estados. En el enfoque clásico o *list scheduling*, la idea es construir una lista ordenada de nodos asignándoles prioridades, y luego repetidamente ejecutar dos pasos hasta obtener un schedule válido: 1) Elegir de la lista el nodo con la mayor prioridad, y 2) Elegir un procesador para alocar el nodo.

El principal problema con los algoritmos de *list scheduling* es que la asignación estática de prioridades no siempre ordena correctamente los nodos, generando schedules muy ineficientes si no puede asignar prioridades precisas. En este sentido un atributo importante de un grafo de tareas que puede usarse para determinar prioridades con precisión se relaciona con el concepto de *camino crítico* (*critical path*, CP), que es el conjunto de nodos y arcos que forman un camino desde un *entry* a un *exit* cuya suma de costos de computación y comunicación es el máximo.

Se sabe que el scheduling multiprocesador para la mayoría de los grafos de tareas con restricciones de precedencia es un problema *NP*-completo en su forma general [116, 206].

Para tratarlo, se hicieron suposiciones simplificadoras teniendo en cuenta la estructura del grafo de tareas del programa y el modelo del sistema de procesadores [54, 133]. De todos modos, el problema es *NP*-completo aún en dos casos simples: 1) scheduling de tareas de tiempo unitario en un número arbitrario de procesadores [138], 2) scheduling de tareas de una o dos unidades de tiempo a dos procesadores [67]. Hay sólo dos casos especiales para los cuales existen algoritmos óptimos de tiempo polinomial: scheduling de grafos de tareas estructurados en árbol con idénticos costos de computación sobre un número arbitrario de procesadores, y scheduling de grafos arbitrarios con idénticos costos de computación en dos procesadores [173, 303]. Sin embargo, aún en estos casos no se asume comunicación entre las tareas del programa paralelo.

Para casos más realistas, un algoritmo de scheduling necesita tratar con un número de temas. Debe explotar el paralelismo identificando la estructura del grafo de tareas, y tomar en cuenta la granularidad de las tareas, computación arbitraria y costos de comunicación. Además para ser de uso práctico, el algoritmo debería tener baja complejidad y ser económico en términos del número de procesadores usados [9, 108]. El tema también es tratado en [19, 65, 120, 225, 66, 361, 95, 99]

El *scheduling dinámico* realiza actividades de planificación concurrentemente en tiempo de corrida y se aplica a problemas dinámicos. Es un enfoque general adecuado para un amplio rango de aplicaciones, puede ajustar la distribución de la carga basado en información del sistema en tiempo de corrida, pero en muchos casos no utiliza información de carga global de los problemas de aplicación. Una estrategia que combine las ventajas del scheduling estático y el dinámico debe ser capaz de generar una carga balanceada sin incurrir en un gran overhead. Esto es posible con técnicas avanzadas de scheduling paralelo. En el *scheduling paralelo* todos los procesadores cooperan para planificar el trabajo.

Entre los algoritmos y modelos relacionados con el problema de scheduling pueden mencionarse: *Task Interaction Graph* (TIG), *Task Precedence Graph* (TPG), *Temporal Task Interaction Graph* (TTIG), *Critical Path* (CP), *Static Scheduling*, *Edge-Zeroing algorithm* (EZ), *Modified Critical Path algorithm* (MCP), *Mobility Directed algorithm* (MD), *Earliest Task First algorithm* (ETF), *Dynamic Level Scheduling algorithm* (DLS), *Dominant Sequence Clustering algorithm* (DSC), *Dynamic Critical Path algorithm*(DCP), *Heterogeneous List Scheduling Heuristic* (HLS), *Heterogeneous Relative Mobility Scheduling algorithm* (HRMS), *LAST Algorithm*, *Duplication Scheduling Heuristics* (DSH), *Vertical Parallel MCP* (VPMCP), *Horizontal Parallel MCP* (HPMCP), *Critical Path Reduction* (CPR), *Parallel Scheduling*, *Tree Walking Algorithm* (TWA), *Cube Walking Algorithm* (CWA), *Mesh Walking Algorithm* (MWA), *Runtime Incremental Parallel Scheduling* (RIPS) [290, 215, 177, 302, 180, 296, 313, 378, 63, 62, 268, 121, 33, 367, 280, 368, 312, 381, 380].

## 5.3 Balance de carga estático

Dado un sistema multiprocesador con una red de interconexión específica, y un algoritmo paralelo compuesto de módulos (tareas) que se comunican, el balance de carga asigna los módulos a los procesadores de manera tal que se minimice el tiempo de ejecución total del algoritmo. El *balance de carga estático* distribuye las tareas al inicio de la computación, y cada una permanece en el procesador durante todo el tiempo de vida del programa. Esto puede realizarse de distintas maneras:

- El diseñador del programa usa *sentido común* para balancear la carga. Por ejemplo, asegurando un número igual de puntos de datos en cada procesador, o decidiendo cuáles tareas son mapeadas en qué procesadores para balancear "groseramente" el sistema. Para ésto necesita saber (o estimar) las variaciones que sufrirá la aplicación cuando se ejecute (lo que no siempre es posible).
- Usar un algoritmo adecuado para encontrar una buena distribución de tareas en procesadores, antes de ejecutar el algoritmo paralelo.

El balance de carga estático brinda poca flexibilidad y requiere un conocimiento *a priori* del problema. Esto presenta dificultades en problemas irregulares o dinámicos, o en algoritmos cuyo tiempo de resolución es altamente dependiente de la carga. Algunos de los casos más comunes en que puede utlizarse balance estático son algoritmos de matrices (tales como factorización LU), gran parte de las computaciones sobre una mesh regular (por ejemplo FFT) o multiplicación matriz-vector sparse.

Existen numerosos estudios sobre el balance de carga estático usando técnicas de teoría de grafos, programación entera, teoría de colas y enfoques heurísticos [371, 324, 291, 314]. En las siguientes Secciones se hará referencia a algunos de los algoritmos más conocidos.

### 5.3.1 Particionamiento de grafo

Si se utiliza un grafo para describir el sistema, puede verse la distribución de carga como un problema de *graph embedding*, esto es, mapear el grafo de la aplicación en el del sistema. Existen varias medidas de costo conocidas para testear la calidad del *embedding* (e.g. carga, dilatación, congestión) [254]. La mayoría de los trabajos consideran grafos estructurados (regulares) como grillas, hipercubos, árboles, etc.

El problema de particionamiento de grafo (*graph partitioning*) puede verse como una relajación del *graph embedding*: describe la tarea de *clustering* de un grafo de aplicación en un número de partes de igual tamaño (tantas como procesadores haya) minimizando el número de arcos que cruzan entre los límites de las particiones (el *cut size*). No se considera un mapeo de clusters a nodos del grafo procesador.

El problema general de $k$-particionamiento puede relajarse a la aplicación recursiva de una cantidad de pasos de bisección donde el grafo se divide en dos partes minimizando el *cut-size*. La principal ventaja del problema de particionamiento en comparación con el de *embedding* es que para muchos grafos populares se conocen bisecciones óptimas, o existen cotas ajustadas [255, 197]. Para el caso general existen heurísticas eficientes que pueden usarse para encontrar soluciones aproximadas al problema de *embedding* [94]. Estas heurísticas se usan en aplicaciones prácticas para mapear datos o procesos en sistemas paralelos.

Las heurísticas de bisección existentes se dividen en *globales* (o de *construcción*) y *locales* (o de *mejora*). Entre las técnicas globales más importantes se encuentran los particionamientos *inercial*, *espectral* y *geométrico* [94, 119, 162]. Entre los métodos locales están la heurística Kernighan-Lin (KL) y la heurística *Helpful-Set* (HS) [94], ambos basados en el principio de búsqueda local con relaciones de vecinaje sofisticadas. Pueden obtenerse mejores resultados combinando heurísticas globales para determinar las soluciones iniciales con métodos locales usados para "ajustar".

### 5.3.2 Algoritmo estático óptimo

Produce un particionamiento óptimo de un programa con estructura de pipeline sobre un arreglo lineal de procesadores. Aunque encuentra la solución óptima, tiene la desventaja de ser computacionalmente caro. El algoritmo consta de 4 pasos:

- Paso 1. Dibujar un grafo en niveles donde cada nivel corresponde a un procesador y cada nodo $< i, j >$ en un nivel corresponde a una subcadena de módulos de $i$ hacia $j$. Si $N$ es el número de procesadores, el nivel 1 contiene los nodos $< 1, j >$ donde $j = 1..m - (N - 1)$. Los niveles $k = 2..n - 1$ contienen los $< i, j >$ donde $i = k..m - (N - k)$, y $j = i..m - (N - k)$. El nivel $n$ contiene los $< i, m >$ donde $i = n..m$.

- Paso 2.

  Un nodo $< i, j >$ en el nivel $k = 1..n-1$ se conecta a todos los nodos $< j+1, q >$ (para algún $q$) en el nivel $k + 1$.

  Todos los $< 1, j >$ en el primer nivel se conectan a un nodo inicial $S$.

  Todos los $< i, m >$ en el nivel final se conectan a un nodo terminal $T$. Cualquier camino que conecta $S$ y $T$ corresponde a una asignación de módulos a procesadores.

- Paso 3. En el nivel $k = 1..n$, cada arco hacia abajo desde el nodo $< i, j >$ es *pesado* con el tiempo requerido por el procesador $k$ para procesar los módulos $i$ hasta $j$ (esto cuenta para el tiempo total de cómputo en el procesador $k$). Los caminos en este grafo representan cada asignación de subcadena contigua posible y el peso del arco

más pesado en un camino corresponde al tiempo requerido por el procesador más cargado. Luego, para tener la asignación óptima se necesita encontrar el camino en el cual el arco más pesado tiene peso mínimo (camino crítico).

- Paso 4. Se encuentra el camino crítico como sigue: los nodos en el primer nivel son etiquetados $L(i)$, y todos los otros $L(i) = \infty$. Comenzando desde arriba y trabajando hacia abajo, se examina cada arco $e$ que conecta un nodo $a$ (above) a un nodo $b$ (below) y se reemplaza $L(b)$ por $min[L(b), max(W(e), L(a))]$, donde $W(e)$ es el peso de $e$. Si varios arcos necesitan ser examinados, se marca el arco que contribuyó al label final $L(b)$. Luego, el camino crítico se encuentra comenzando por $T$ y siguiendo los arcos marcados.

Si la cantidad de tareas es $m$, el número de nodos por nivel es $O(m^2)$ y hay $n$ niveles, de modo que se tienen $O(m^2n)$ nodos en el grafo. Además, el número de arcos que salen de un nodo es a lo sumo $m$, por lo que la complejidad en tiempo es $O(m^3n)$.

### 5.3.3 Método *Binary Dissection*

Brinda soluciones cercanas al óptimo y computacionalmente eficientes. La cadena de $m$ módulos se divide en dos secciones tal que la diferencia entre las sumas de los tiempos de ejecución en cada sección es un mínimo. Las dos secciones son subdivididas recursivamente tantas veces como se quiera.

La cantidad de piezas en que puede particionarse la cadena es $2^k$, donde $k$ es la profundidad del particionado (número de niveles). Luego, el algoritmo es útil para problemas en que el número de procesadores es potencia de 2. El tiempo requerido es $O(m \log_2 N)$ ya que no puede haber más de $O(\log_2 N)$ niveles de particionado y cada nivel requiere a lo sumo un acceso al peso de cada módulo.

### 5.3.4 Algoritmos de bisección recursiva

Se utilizan para dividir grillas de forma que la carga computacional se distribuya equitativamente, minimizando el costo en las comunicaciones. Tiene la ventaja de que localidad en el espacio se corresponde con la localidad en memoria, lo cual ayuda en la performance en las arquitecturas basadas en *caches*.

Los algoritmos de bisección recursiva comienzan dividiendo el dominio original en dos subdominios. Luego, cada subdominio se divide en dos, y así hasta obtener la cantidad de subdominios requerida. Esta estrategia recursiva permite al algoritmo de división de tareas ejecutarse en paralelo.

La diferencia entre los algoritmos de bisección radica en cómo ven a la estructura (un grafo, una mesh, etc.) y en la decisión de cuándo y cómo llevar a cabo la división del dominio. Algunos ejemplos son: bisección recursiva de grafo, bisección recursiva ortogonal, bisección recursiva desbalanceada, ERB (*Eigenvalue Recursive Bisection*).

### 5.3.5 Algoritmo Greedy

Sean $A_{sum}$ el peso total de todos los módulos, $\hat{A}_{max}$ el peso del módulo de mayor peso, y $A^{opt}$ el peso de la subcadena más pesada en la partición óptima. Si $N$ es el número de procesadores, puede mostrarse que $A^{opt}$ siempre está en el rango $(A_{sum}/N) \leq A^{opt} \leq (A_{sum}/N) + \hat{A}_{max}$. Esto forma la base del algoritmo greedy.

Se elige un peso de prueba $A$ en el rango anterior y se trata de particionar la cadena de módulos en subcadenas tal que el peso de cada subcadena sea menor o igual que $A$. Si se encuentra una partición se la llama *partición greedy*. Si no existe una partición greedy entonces se puede reducir el peso y tratar y encontrar otra partición greedy más precisa; en otro caso, el peso puede incrementarse. La selección del peso de prueba $A$ se encuentra mediante una búsqueda binaria en el rango $(A_{sum}/N) \leq A \leq (A_{sum}/N) + \hat{A}_{max}$.

Una visión alternativa es la siguiente: comenzando por un vértice con el menor grado, marcar sus vecinos y luego los vecinos de éstos. Los primeros $n/p$ vértices marcados se toman para formar un subdominio y el procedimiento se aplica al grafo restante hasta que todos los vértices se marquen. Este algoritmo tiene una complejidad $O(n)$.

### 5.3.6 Selección al azar

Una de las formas más simples de alocar tareas en procesadores consiste en elegir un procesador al azar y darle la tarea. Si el número de tareas es grande, cabe esperar estadísticamente que cada procesador reciba una carga similar. Las mayores ventajas de este método son su bajo costo y la alta escalabilidad. La principal desventaja es que la comunicación con tareas que están fuera del procesador puede ser una necesidad para casi todas y este algoritmo no lo tiene en cuenta. Además, sólo se obtiene un balance de carga aceptable cuando la cantidad de tareas es muy superior a la cantidad de procesadores.

Esta estrategia tiende a ser más efectiva cuando hay poca comunicación entre tareas y cuando los patrones de comunicación son infrecuentes. En otros casos resulta en un mayor overhead de comunicación que otros métodos.

### 5.3.7 Round Robin

Los métodos *round robin* se reducen a asignar a cada procesador $P_i$ una tarea cada $p$ asignaciones de tarea. Por ejemplo, con quince tareas para distribuir (1, ..., 15) y cinco procesadores (1, ..., 5), a $P_2$ le tocarían las tareas 2, 7 y 12.

Si bien se trata de un algoritmo simple, no toma en cuenta el aspecto de la localidad de las comnicaciones. Una alternativa que puede resultar útil en algunos casos y que utilizan algunos algoritmos es asignar bloques en lugar de tareas individuales.

### 5.3.8 Algoritmos de optimización global

Dado que el partionamiento de grafo es un problema de optimización global, se han explorado una cantidad de algoritmos de este tipo. Entre ellos se encuentran *simulated annealing* [360, 244, 351, 247], algoritmos genéticos (GA) [245, 56, 2] y algoritmos evolutivos. Estos métodos pueden utilizarse también en balance dinámico, y se describen más detalladamente en la Sección 5.4.12.

Estos algoritmos en su forma secuencial tienden a tener un tiempo de ejecución significativo (y en algunos casos mayor uso de memoria) comparados con otros algoritmos de particionamiento. Pero, son fáciles de paralelizar y también tienen el potencial de brindar particiones de mayor calidad. En particular, pueden ser útiles en el ajuste (*tunig*) fino de una partición existente creada por un algoritmo más rápido. Además, generalmente se utiliza un modelo más complicado y preciso, en la forma de una función de costo que toma en cuenta el efecto del costo de comunicación y el balance de carga [353].

## 5.4 Balance de carga dinámico

En muchas ocasiones no es posible conocer *a priori* la evolución de la aplicación, por lo que se debe utilizar alguna política de balance de carga dinámico con los objetivos de minimizar el tiempo de ejecución promedio y mejorar el uso de los PEs [59, 179]. Se realizan etapas de balanceo durante la ejecución de la aplicación; en general estos métodos requieren alguna forma de mantener una visión global del sistema (a distintos niveles de acuerdo al algoritmo) y algún mecanismo de negociación para la migración de procesos y/o datos [371].

Si bien el balance dinámico tiene el potencial de mejorar la performance global de la aplicación redistribuyendo la carga de trabajo entre los elementos de procesamiento, esta actividad se realiza a expensas de computación útil, produce overhead de comunicación y requiere espacio en memoria para mantener la información [236]. Además, los overhads en que incurren estos métodos en muchos casos dependen de la topología [240].

Para justificar el uso de estrategias de balance de carga, la precisión de cada decisión de balanceo debe pesarse contra la cantidad de procesamiento y comunicación agregados por este proceso. Deben resolverse temas tales como cuándo invocar un balanceo, quién toma las decisiones de balance y de acuerdo a qué información, y cómo manejar las migraciones. Las diferentes respuestas a estas preguntas abren un amplio espacio de diseños posibles para métodos de balance de carga dinámicos.

Una primera diferenciación entre los métodos de balance de carga dinámicos es la que los clasifica en *centralizados* y *descentralizados* (distribuidos o locales). Los primeros requieren una vista completa del sistema, mientras los descentralizados realizan el balance mediante información obtenida sólo de procesadores vecinos.

En general, los algoritmos descentralizados tienen un costo en tiempos de ejecución y comunicación bastante reducido, lo que los hace adecuados en situaciones donde la carga varía en forma constante. Pero en muchos casos no resultan tan buenos como los centralizados, y pueden ser lentos a la hora de adaptarse a cambios importantes en las cargas de los procesadores. Por ejemplo, si de repente un procesador estuviera cargado en exceso, se requerirían muchos balances locales para *esfumar* la carga a otros procesadores.

Otra clasificación, de acuerdo a cómo tiene lugar la migración, divide a los métodos en *iterativos* y *directos*. En los primeros, los procesos migran "iterativamente" - esto es, un paso (hacia un procesador vecino) por vez -, cada paso de acuerdo a una decisión local hecha por el procesador intermedio. En cambio, un procesador que ejecuta un procedimiento directo tomaría decisiones sobre los destinos finales de los procesos locales que quiere migrar. Los métodos directos, a causa de su necesidad de *matchear* eficientemente emisores y receptores de cargas de trabajo son más apropiados para sistemas equipados con un mecanismo de *broadcast* o un monitor centralizado. Por otro lado, los iterativos, caracterizados por su comportamiento "una-decisión-a-la-vez", funcionan mejor en multicomputadores basados en una red de comunicaciones punto a punto.

Los métodos iterativos se basan en aproximaciones sucesivas a una distribución de carga global óptima, y por lo tanto en cada iteración sólo necesitan tratar con la dirección de la migración de la carga. Algunos seleccionan una única dirección (uno de sus vecinos más cercanos) mientras otros consideran todas las direcciones (todos sus vecinos más cercanos). Estos métodos pueden categorizarse en *determinísticos* (funcionan de acuerdo a ciertas reglas predefinidas por las cuales a qué vecino se le transfiere la carga extra y cuánto se le transfiere depende de ciertos parámetros tales como los estados de los procesos vecinos) y *estocásticos* (las cargas se redistribuyen de manera randomizada, sujeto al objetivo del balance). Entre los primeros se encuentran, entre otros, los métodos de difusión, *dimension exchange* y gradiente. Como estocásticos pueden mencionarse la alocación randomizada, *simulated annealing*, algoritmos genéticos y redes neuronales.

En general, la performance es una medida absoluta descripta en términos de tiempo de respuesta, utilización, o cualquier función objetivo especificada. Las actividades de

todas las políticas de LB introducen un overhead limitado, definido como el porcentaje de recursos sustraídos por la política de aquellos disponibles para las aplicaciones. Una posibilidad es analizar la *performance normalizada* y el *tiempo de estabilización*.

La performance normalizada o *tiempo de respuesta normalizado*(NRT) determina la efectividad de la estrategia de balance. Es una métrica comprensiva; toma en cuenta el nivel de desbalance de carga inicial así como los overheads de balanceo. Formalmente: $NRT = \frac{T_{nolb}-T_{bal}}{T_{nolb}-T_{opt}}$, donde $T_{nolb}$ es el tiempo para completar el trabajo en una red multiprocesador sin balanceo de carga, $T_{opt}$ es el tiempo para completar la tarea en un procesador dividido por el número de procesadores en la red, y $T_{bal}$ es el tiempo para completar el trabajo en la red con balance de carga. $NRT = 0$ si la estrategia no es efectiva y $NRT = 1$ si la estrategia es efectiva. Cuando el tiempo con balance de carga se aproxima al óptimo, entonces $NRT$ se acerca a 1. Por otro lado, si el balanceo es pobre y no mejora respecto del caso sin balanceo de carga, entonces $NRT$ se acerca a 0.

El tiempo de estabilización o tiempo de balanceo de carga indica cuánto le toma a la red lograr un estado balanceado donde no se requieran más transferencias de tareas. Un tiempo de estabilización bajo no necesariamente indica una estrategia eficiente. Podría además indicar que, a causa de información inadecuada, la red balanceada es subóptima. Tal red aún puede tener una carga distribuida no equitativamente aunque el desbalance es insuficiente para disparar las actividades de balanceo.

### 5.4.1 Estrategias iniciadas por el emisor

En estos esquemas la división del trabajo es iniciada por el emisor: la generación de subtareas es independiente de los pedidos de trabajo desde los procesadores ociosos [210, 240, 86, 115, 283, 311]. Un procesador sobrecargado (*emisor*) tratando de enviar una tarea a un procesador subcargado (*receptor*) inicia la distribución de carga.

Eager *et al.* [97] propusieron tres estrategias totalmente distribuidas, que difieren en la política usada para ubicar los procesadores que intercambian tarea. Sin embargo, las tres tienen desventajas como la falta de un mecanismo que asegure que el procesador subcargado elegido está a una distancia moderada del emisor, y aún que el elegido sea el mejor candidato. En algunos casos esto puede no asegurar consistencia en la comparación de performance con otros modelos como el de gradiente o los de transferencias iniciadas por el receptor. Esto puede subsanarse si se usa el modelo propuesto por Willebeek-LeMair y Anthony Reeves que usa sólo información de estado del vecino inmediato [358]

#### Balance de carga *Single Level* (SL)

Este esquema balancea la carga dividiendo la tarea en un gran número de subtareas tal que cada procesador es responsable de más de una subtarea. Estadísticamente esto

asegura que el trabajo total en cada procesador es aproximadamente el mismo. Un procesador MANAGER genera un número específico de subtareas y se las da de a una a los procesadores que lo requieren. Dado que MANAGER tiene que generar subtareas lo suficientemente rápido para mantener ocupados a los procesadores, la creación de subtareas forma un cuello de botella que previene la escalabilidad del esquema. En [210, 115] se analiza el tema de la escalabilidad para el caso de subtareas de igual o distinto tamaño.

### Balance de carga *Multi Level* (ML)

Trata de evitar el cuello de botella de SL mediante múltiples niveles. Los procesadores son organizados en forma de árbol $m$-ario de profundidad $l$. El procesador raíz divide la tarea en *super-subtareas* y las distribuye a sus sucesores bajo demanda. Estos, a su vez, subdividen las *super-subtareas* en subtareas y las distribuyen a sus sucesores a pedido. Los nodos hoja repetidamente piden trabajo a sus padres tan rápido como terminan lo asignado. Un procesador hoja es alocado a otro generador de subtareas cuando el suyo queda sin trabajo. Para $l = 1$, ML es idéntico a SL. [210] presenta un estudio de escalabilidad e isoeficiencia de esta estrategia.

### Alocación randomizada

Existen numerosas técnicas usando alocación randomizada presentadas en el contexto de búsquedas *depth-first-search*(DFS) o árboles de espacio de estado [283, 311, 293]. En el DFS de árboles, la expansión de un nodo corresponde a realizar una cierta cantidad de computación útil y generación de nodos sucesores, los cuales pueden ser tratados como subtareas.

En la estrategia propuesta por Shu y Kale [311], cada vez que un nodo es expandido, todos los sucesores generados se asignan a procesadores elegidos al azar, lo que daría algún grado de balance. Este esquema tiene algunas dificultades de implementación y es aplicable sólo cuando el costo de cómputo asociado a cada nodo es mucho mayor que el costo de comunicación. En algunos problemas pácticos, es mucho más "barato" construir incrementalmente el estado asociado con cada nodo en lugar de copiar y/o crear el nuevo nodo desde *scratch* (introduciendo ineficiencia adicional). Además, la memoria necesaria en un procesador es potencialmente ilimitada, ya que se le puede requerir almacenar un número arbitrariamente grande de piezas de trabajo durante la ejecución.

Ranade [283] presentó una variante para la ejecución sobre redes *butterfly* o hipercubos, usando un algoritmo dinámico para embeber los nodos de un árbol binario de búsqueda en una red *butterfly*. El algoritmo particiona el trabajo en cada nivel en dos partes y las envía a los dos hijos en la red, asegurando un grado de balance.

Otro caso de alocación randomizada es el conocido como *balls into bins games*, con-

sistente en asignar jobs entrantes (*balls*) a procesadores (*bins*) equitativamente. Una posibilidad es chequear la carga de un número constante de *bins* elegidos al azar antes de asignar *balls*; otro esquema consiste en ubicar un número constante de copias de *balls* en diferentes *bins* elegidos al azar [279, 253, 58].

### 5.4.2 Estrategias iniciadas por el receptor

En estos esquemas, cuando un procesador (*receptor*) se queda sin trabajo, genera un pedido. La selección del destino de este pedido es lo que diferencia a las estrategias; la elección podría ser tal como minimizar el número total de pedidos y transferencias así como el desbalance entre procesadores [210, 240].

En cualquier instante de tiempo, algunos de los procesadores están ocupados (tienen trabajo) y otros están ociosos (tratando de obtener trabajo). Un procesador ocioso elige otro como destino y le envía un pedido de trabajo; si recibe algún trabajo desde el procesador destino, se vuelve ocupado. Si recibe un mensaje de *reject* (porque el destino no tiene trabajo disponible) elige otro y le envía el pedido. Esto se realiza hasta que obtiene trabajo o todos se vuelven ociosos. Cuando un procesador con trabajo recibe un pedido, particiona en dos su trabajo y le da una de las partes al solicitante; si la cantidad de trabajo que tiene es demasiado poca envía un *reject*.

A continuación se describen algunas de las estrategias iniciadas por el receptor más conocidas. En [210] se presenta un análisis de escalabilidad de las mismas sobre las arquitecturas hipercubo, mesh y red de workstations.

#### Round Robin Asincrónico (ARR)

Cada procesador mantiene una variable independiente $target$. Cuando se queda sin trabajo, lee su valor de $target$, envía un pedido al procesador con ese identificador, e incrementa $target$ (módulo $P$). Inicialmente $target(p) = ((p+1) \bmod P)$, donde $p$ es el número identificador del procesador. Cada uno puede generar pedidos independientemente del resto. En un esquema simple y sin cuellos de botella, aunque podrían enviarse casi al mismo tiempo muchos pedidos al mismo procesador, lo que no es deseable.

#### Nearest Neighbor (NN)

Cuando un procesador se queda sin trabajo envía un pedido a sus vecinos inmediatos de manera *round robin*. Por ejemplo, en un hipercubo, un procesador sólo envia pedidos a sus $\log P$ vecinos. En redes donde la distancia entre todos los pares de procesadores es la misma, este esquema es idéntico a ARR. Asegura localidad de comunicación para

los pedidos y transferencias, aunque le toma mayor tiempo distribuir las concentraciones localizadas de trabajo.

### Round Robin Global (GRR)

Una variable global TARGET se almacena en el procesador 0. Cada vez que alguno necesita trabajo, pide y toma el valor de TARGET, y el procesador 0 incrementa el valor de la variable en 1 (módulo $P$) antes de responder otro pedido. Luego, el que necesita trabajo envía un pedido al procesador cuyo número coincide con el valor leído de TARGET. Esto distribuye equitativamente el trabajo, aunque puede provocar contención en el acceso a la variable global.

### GRR con combinación de mensajes (GRR-M)

Es una versión modificada de GRR que evita contención en el acceso a TARGET. Todo los pedidos para leer el valor en el procesador 0 se combinan en procesadores intermedios. Así, el número total de pedidos a manejar se reduce. Esta técnica de realizar operaciones de incremento atómicas sobre una variable compartida es una implementación software de la operación *fetch-and-and* de [137]. En [210] puede encontrarse un descripción de este esquema sobre un hipercubo.

### Random Polling (RP)

Es la estrategia de balance de carga más simple donde un procesador cada vez que se queda sin trabajo le pide a otro elegido al azar. La probabilidad de selección de cada procesador es la misma.

### Balance basado en scheduler (SB)

Fue propuesto en [271] en el contexto de búsqueda *depth first* para generación de tests en aplicaciones CAD VLSI. Un procesador es designado como *scheduler* y mantiene una cola llamada DONOR con todos los posibles procesadores que pueden donar trabajo. Inicialmente DONOR contiene sólo un procesador que tiene todo el trabajo; cada vez que alguno se vuelve ocioso, envía un pedido al scheduler, quien borra a este procesador de la cola DONOR y hace *polling* de la cola en forma *round robin* hasta que toma trabajo de uno de los procesadores. En este punto, el receptor es ubicado al final de la cola y el trabajo recibido por el scheduler es redirigido solicitante.

En este esquema, como en GRR, los pedidos sucesivos son enviados a distintos procesadores. Pero, aquí un pedido nunca es enviado a alguien que se sabe que no tiene trabajo. La performance puede degradarse por el hecho de que todos los mensajes, incluso los que llevan el trabajo, son ruteados a través del scheduler. Esto puede modificarse de modo que el polling sea generado por el scheduler pero el trabajo sea transferido directamente al que lo solicita. Si el consultado está ocioso, retorna un mensaje de *reject* al scheduler; al recibir un reject el scheduler toma al próximo de DONOR y genera otro poll. Este esquema modificado logra mejor performance que el original [210].

### 5.4.3 Estrategias basadas en predicción

Intentan balancear la carga como resultado de predicciones sobre los requerimientos de un proceso. La estrategia propuesta por Goswami *et al.* [136] demostró predicción de los requerimientos de CPU, memoria y entrada/salida de un proceso, antes de su ejecución, usando un método estadístico de reconocimiento de patrones. Sin embargo, aunque los valores predichos están cerca de los reales, la estrategia incurre en overheads de cómputo significativos y usa números identificadores de tareas dependientes de la red para tabular los posibles resultados.

Otros autores propusieron una estrategia que usa probabilidades de transferencia de tareas para predecir los requerimientos de carga de un procesador [106]. Los modelos de probabilidad son más realistas, ya que capturan características de tiempo variable en el scheduling distribuido de aplicaciones. Otra ventaja es que la red puede estimar la carga de un procesador en cualquier momento sin preguntarle a ese procesador.

Esta estrategia usa el tiempo de servicio $S_i(t)$ como el índice de carga para realizar el balance dinámico. Cada procesador estima su propio tiempo de servicio para el próximo intervalo de tiempo y hace broadcast del mismo a los otros. Durante un intervalo $\Delta t$, la red puede estimar el tiempo de servicio $S_i(t)$ registrando el tiempo total usado por el procesador $i$ sirviendo tareas, y el número de salidas de tareas completadas en ese intervalo.

En un tiempo $t$ específico, $S_i(t) = \frac{\Delta t}{d_i(t)}$, donde $S_i(t)$ es el tiempo de servicio por tareas, y $d_i(t)$ es el número total de partidas de tareas en $\Delta t$. Cada uno de los $n$ procesadores distribuye esta información a los otros y computa su tiempo medio de servicio para la red $S_m(t) = \frac{S_1(t)+...+S_n(t)}{n}$. Luego, cada procesador determina el estado de su carga y la del resto: si $S_i(t) > S_m(t)$ está muy cargado, y en caso contrario está poco cargado. El paso siguiente involucra determinar $W(t)$, el cociente de tiempo de servicio de exceso y el tiempo de servicio medio, en cada procesador muy cargado $i$: $W_i(t) = \frac{S_i(t)-S_m(t)}{S_i(t)}$.

Finalmente, cada procesador $i$ computa y mantiene una lista de probabilidades de trasferencia de tareas entre él y todos los otros procesadores sobrecargados $j$: $p_{ij} =$

$\left(\frac{\sum_{k=1}^{L} S_k(t) - S_j(t)}{\sum_{k=1}^{L} S_k(t)}\right) W_i(t)$ donde $L$ es el número de procesadores poco cargados. El procesador sobrecargado elige el poco cargado con la mayor probabilidad de transferencia. El número de tareas a transferir es proporcional a $W_i(t)$.

En el Capítulo 7 se presenta un método de ordenación con balance de carga dinámico que utiliza un estimador del trabajo para predecir la carga en cada procesador.

### 5.4.4 Métodos difusivos

En los algoritmos de tipo *nearest neighbor* los procesadores toman decisiones basados en información local de manera descentralizada y manejan las migraciones con sus vecinos inmediatos [372, 3, 194, 210, 358, 369, 374]. Dado que sólo esparcen trabajo localmente, son algoritmos escalables para operar en máquinas masivamente paralelas de cualquier tamaño, y tienden a preservar la localidad de comunicación de las computaciones subyacentes.

En general son ejecutados iterativamente, con la expectativa de que sucesivas invocaciones de balanceo local eventualmente llevarían a un estado balanceado. Esto es: realizan aproximaciones sucesivas a una distribución uniforme global, y en cada operación sólo tratan con la dirección de la migración de la carga y el tema de cómo prorratear los excesos de trabajo. Los métodos difusivos son adecuados para implementar en una arquitectura de comunicación básica *all port*, que permite a un procesador intercambiar mensajes con todos sus vecinos directos simultáneamente en un paso de comunicación.

En la literatura, los métodos difusivos y de *dimension exchange* recibieron gran atención tanto teórica como experimental. El método de difusión fue modelizado inicialmente usando teoría de sistemas lineales por Cybenko [84] y Bertsekas y Tsitsiklis [38]. Cybenko mostró que el método eventualmente coaccionará a cualquier distribución de carga inicial a una distribución uniforme global en situaciones estáticas en que no generan o consumen cargas durante el balanceo, y presentó una cota asintótica de la varianza de cualquier distribución durante el balanceo en la situación dinámica. Resultados similares obtuvo Boillat [45], quien también probó que el método converge a un estado balanceado global en tiempo polinómico. Hong *et al.* [171] y Qian y Yang [277] presentaron una cota constante para la varianza de la distribución de la carga cuando se aplica el método a estructuras específicas. El método de difusión está caracterizado por un parámetro que determina la porción de carga de exceso a ser difundida. Xu y Lau analizaron los efectos del parámetro sobre la eficiencia del método, y derivaron valores óptimos para mesh y toro [372].

Los beneficios del método de difusión se mostraron en el contexto de computaciones distribuidas de algoritmos *branch-and-bound* [358, 375]. También, Willebeek-LeMair y Reeves [358] compararon los resultados de difusión y *Dimension Exchange* (DE) en la computación distribuida de algoritmos *branch-and-bound* en un hipercubo, concluyendo

(en concordancia con Cybenko) que el speedup de DE es mejor que el de difusión.

Pueden modelizarse varios algoritmos de balance de carga con diferentes grados de vecinaje y coordinación: desde un algoritmo local en que la coordinación se limita a pares de nodos a otros menos locales que coordinan acciones en dominios más grandes. Estas políticas se inspiraron en el fenómeno físico de la *difusión*, que balancea distribuciones escalares no homogéneas sólo sobre la base de estados locales y moviendo items localmente en la dirección sugerida por un objetivo de minimización de energía. Una política de balance de carga distribuida puede balancear la carga sólo sobre la base de información de carga local y migrando items de ejecución (procesos, *threads* u objetos activos) en las direcciones de carga decreciente [72, 73].

Una política de balance de carga es difusiva cuando:

- Se basa en *componentes de decisión replicados*, cada uno con igual comportamiento y capaz de realizar actividad autónoma y asincrónica.
- El *objetivo de balance de carga se persigue localmente*: el alcance de las acciones de cada componente de decisión está limitada a un área local del sistema (*dominio de localidad*); cada componente trata de balancear en su dominio como si fuera todo el sistema, sólo sobre la base de la *información de carga* en su dominio.
- El dominio de cada componente su superpone parcialmente con el controlado por al menos otra componente; la unión de los dominios da una *cobertura total* del sistema.

Pueden derivarse distintas políticas a partir del esquema de difusión, cuya implementación debe ocuparse de varios temas en las fases de decisión local que preceden a las acciones de balanceo (antes de las migraciones). En particular, para cada nodo:

- Fase de *triggering*: la componente de decisión identifica las condiciones que inician las sucesivas fases de balance de carga.
- Fase de *identificación de estado*: determina si los nodos de su dominio se encuentran en un estado que requiere acciones de balanceo.
- Fase de *ubicación*: identifica, en el dominio, los nodos subcargados (receptores) y los sobrecargados (emisores) que necesitan acciones de balanceo.
- Fase de *selección*: elige qué items mover del emisor al receptor.

La actividad de decisión de balance de carga se completa con las acciones de balanceo necesarias, es decir, migración de items de carga.

Con respecto a la fase de triggering, las políticas pueden ser periódicas o *event-driven*. Las primeras operan a intervalos predefinidos; producen overhead predecible, aunque el intervalo óptimo depende de la aplicación. Las event-driven son disparadas por cambios en la carga local y por recepción de información de carga actualizada (adaptan sus actividades a la evolución de la aplicación); a causa del alto overhead si hay demasiados eventos de triggering, una pequeña variación de carga no causa transmisión de nueva información.

En la fase de identificación de estado no es útil comenzar a trabajar por cualquier desbalance: cuando se está en una situación cercana al balance podrían gastarse recursos sin lograr progreso significativo. Esto se debe al costo no despreciable de los mecanismos de realocación y a la granularidad del item (los items pueden ser demasiado grandes para permitir, con su movimiento, una reducción del desbalance). La política debe inhibir tomar acciones si el desbalance está por debajo de un *threshold*. En el caso de políticas adaptivas, este valor es calculado dinámicamente como función de la carga corriente.

Dependiendo del rol de los nodos en la fase de ubicación, la política de balance se define como iniciada por el emisor, iniciada por el receptor o iniciada simétricamente. Las políticas difusivas no son tan estrictas respecto a la iniciativa: una vez que los estados de carga de los nodos en un dominio fueron identificados, los *partners* se relacionan uno con otro cualquiera sea el nodo que inició la acción. La necesidad de acelerar las actividades sugiere evitar largas fases de negociación.

El tema importante de la fase de selección es la granularidad de la carga: dado que la carga no es divisible indefinidamente, la elección de qué items migrar debe tomar en cuenta la carga de cada ítem. En el caso de granularidad de carga muy gruesa (y con la influencia del threshold elegido), la política detiene su actividad. Esta condición es esecial para lograr acciones efectivas: cuando la carga migra del emisor al receptor la política debe evitar mover un exceso de carga que revirtiera los roles de los nodos involucrados. Otros requerimientos pueden restringir aún más los ítems a migrar.

Dado que las políticas difusivas llevan a cabo acciones que pueden superponerse en tiempo y espacio sobre dominios conectados, la implementación debe garantizar un principio de *serialización* para garantizar consistencia: un nodo no puede participar en acciones de balanceo concurrentes en dominios superpuestos.

A continuación se presentan algunas políticas difusivas con diferentes definiciones de dominio de localidad. En cada una se asume una entidad replicada en cada nodo del sistema (el *Allocation Manager*, AM) que se encarga de la implementación local de la política y de garantizar la serialización coordinando con los otros AMs [73].

### Política Direct Neighbor (DN)

Elige el mínimo tamaño de dominio: cada dominio de localidad consta sólo de dos nodos conectados por un link físico. Dado un nodo $N1$, el arribo de información de carga

actualizada desde su vecino ($N2$) dispara la actividad de LB en el dominio $N1 - N2$. El AM de $N1$ compara la carga de $N2$ con la suya. Si la carga de $N1$ excede la de $N2$ en más de un threshold (calculado como porcentaje de la carga local), el AM de $N1$ trata de balancear el trabajo de $N1$ y $N2$.

Las fases de identificación de estado y de ubicación se superponen: la primera compara la carga de los nodos e inmediatamente establece la pareja emisor-receptor. La política no incluye ninguna orden para determinar la nueva posición: sólo garantiza la serialización en dominios superpuestos. Por ejemplo, $N3$ en el dominio $N3 - N1$ no puede comandar ninguna migración hacia $N1$ si éste ya está involucrado en una acción en el dominio $N1 - N2$; cuando se completa, $N1$ puede aceptar una migración.

La fase de selección determina que la migración de items no invierte los roles emisor-receptor, lo que garantiza que las acciones de balanceo siempre van en la dirección de balancear ese dominio.

### Política Average Neighborhood (AN)

Agranda la dimensión de los dominios de localidad: un dominio está constituido por un nodo y todos sus vecinos directos. Si $K$ es el número de vecinos directos del nodo $N$, entonces $N$ pertenece a lo sumo a $K+1$ dominios: en uno de ellos es el centro (*Maestro*) y vecino directo de todos los otros; y es un nodo periférico (*Esclavo*) en los otros dominios a que pertenece.

El AM de $N$ se encarga de las decisiones de balanceo para el dominio $D$ del cual es el Maestro, ya que es el único en $D$ que puede acceder y controlar a los otros. El AM de $N$ mantiene la información de carga de cada nodo de $D$ y computa dinámicamente la carga promedio del dominio y dos thresholds ($T_{emisor}$ y $T_{receptor}$) centrados en él. Un nodo se clasifica como emisor en $D$ si su carga excede $T_{emisor}$, como receptor si su carga está por debajo de $T_{receptor}$, y como nodo neutral en otro caso.

Cada vez que el Maestro $N$ recibe información actualizada, el trigger fuerza al AM a recomputar la carga promedio y los thresholds, y a evaluar la situación de carga en $D$. El AM de $N$ trata de llevar la carga de todos en $D$ (no sólo de sí mismo) tan cerca como sea posible del promedio y sin realizar migraciones que pudieran invertir el rol de los nodos.

### Política Average Extended Neighborhood

AN define los dominios de localidad sobre la base de la relación de vecinaje directo con el Maestro; una extensión natural es agrandar el tamaño de los dominios. Un dominio puede definirse como el conjunto de nodos cuya distancia desde el Maestro es menor o igual que $d$. Con $d = 1$, la estrategia es AN, con $d = 2$ la estrategia define un vecinaje

promedio de segundo nivel (AN-2), y así sucesivamente (estrategia AN-d).

Aunque es difusiva, la implementación de AN-d, con $d > 1$, introduce más problemas que AN: requiere comunicaciones entre nodos no vecinos y el Maestro puede tener problemas para evaluar la carga promedio del dominio. Por esto, muchos estudios se limitan a AN-2.

### Política Direct Neighbor Repeated (DNR)

Explota más información que DN durante la fase de ubicación, manteniendo las mismas fases de triggering, identificación de estado y selección. En DN, una vez que se estableció una pareja $N1 - N2$, la carga puede moverse, desde el emisor $N1$, sólo al receptor $N2$. A su turno, $N2$ puede tener un vecino aún menos cargado ($N3$), que pertenece a un dominio DN distinto. DNR identifica estas situaciones y permite a los nodos receptor forwardear directamente la carga a nodos más subcargados. La migración se detiene sólo cuando no hay más movimientos útiles.

DNR agranda la localidad de las acciones de LB brindando la posibilidad de mover items en saltos sucesivos como una migración única. Desde un punto de vista de implementación, el forward de la carga es sencillo: cuando una pareja direct-neighbor es establecida y un nodo $Ny$ está esperando carga, $Ny$ sabe si existe un nodo menos cargado en su vecindario. En este caso, $Ny$ actúa como un *forwarder* hacia el vecino subcargado.

Esta política aún es difusiva. La diferencia con las anteriores es la definición dinámica de dominios de localidad.

### Comentarios

Corradi *et al.* en [73] presentan una comparación de las políticas difusivas. Allí muestran que todas están cercanas al caso ideal de LB para situaciones *lentamente* dinámicas, aunque DN Y AN tienen un peor NRT causado por un mayor número de migraciones. AN-2 mejora el NRT gracias a agrandar sus dominios de localidad. DNR logra el mejor NRT obteniendo la mejor calidad de balanceo con el menor esfuerzo de migración.

En situaciones altamente dinámicas, NRT decrece para todas las políticas, debido a la peor calidad del balanceo y el mayor esfuerzo de LB. Para AN-2 y DNR, los NRT se vuelven negativos, significando que la aplicación de las políticas no es beneficiosa. Para esta dinamicidad, sólo AN y DN son efectivas.

Versiones modificadas de métodos difusivos para acelerar la convergencia se presentan en [92, 123, 174, 195].

### 5.4.5 Balance jerárquico

La estrategia HBM (*Hierarchical Balancing Method*) organiza un sistema multicomputador en una jerarquía de dominios de balanceo, descentralizando el proceso [358]. Se designan procesadores especiales para controlar las operaciones de balance en distintos niveles de la jerarquía. Por ejemplo, en una organización tipo árbol binario, los procesadores a cargo del LB en un nivel $l_i$ reciben información de carga de los dos dominios del nivel inferior ($l_{i-1}$). El balance global se logra ascendiendo el árbol y balanceando la carga entre dominios adyacentes en cada nivel de la jerarquía.

El procedimiento es asincrónico, y el balanceo es invocado dentro de un dominio cada vez que el controlador del mismo detecta un desbalance. Para una configuración jerárquica binaria el tamaño de los dominios se duplican de un nivel al siguiente. La estructura en árbol minimiza el overhead de comunicación y puede escalar para grandes sistemas.

### 5.4.6 Dimension Exchange

El método *Dimension Exchange* (DE) pertenece a la clase de algoritmos *nearest neighbor*, y es adecuado para implementar en una arquitectura de comunicación básica *one port*, donde un procesador puede intercambiar mensajes con a lo sumo un vecino directo a la vez. Fue diseñado y estudiado intensamente para máquinas paralelas con estructura de hipercubo, donde el balanceo procede iterativamente en dimensiones [285, 310]. En cada dimensión, un procesador balancea su carga con su vecino que pertenece a esa dimensión. Una "barrida" (*sweep*) del proceso iterativo corresponde a ir a través de las dimensiones del hipercubo una vez. Dado que el conjunto de vecinos corresponde exactamente a las dimensiones del hipercubo, el procesador habrá comparado e intercambiado carga con cada uno de sus vecinos después de un *sweep*.

Cybenko mostró que, independientemente del orden en que se consideren las dimensiones, el método logra una distribución uniforme a partir de cualquier distribución inicial después de una ronda de operaciones de balanceo [84]. Además mostró la superioridad de DE sobre difusión en hipercubos ya que este último logra menor varianza de la distribución de carga en situaciones no quietas. Este resultado teórico fue sustentado por el experimento de Willebeek-LeMair y Reeves [357] sobre algoritmos *branch-and-bound*. Antes del balanceo se requiere una sincronización global de modo que todos los procesadores estén listos para la ejecución del mismo.

EL método no se limita a hipercubos. Hosseini *et al.* lo aplicaron a estructuras arbitrarias basándose en *edge-coloring* [172], donde los arcos de un grafo son coloreados con un número mínimo de colores tal que no hay dos arcos adyacentes con el mismo color, y una *dimensión* se define como el conjunto de arcos del mismo color. Mostraron que dada una estructura arbitraria, DE converge eventualmente a una distribución uniforme.

DE es similar a HBM en que pequeños dominios son balanceados primero y luego combinados para formar mayores dominios hasta que todo el sistema está balanceado. Difieren en que DE es sincronizado. El método DE fue experimentado en particionamiento paralelo de grafos [93] y remapeo periódico de computaciones *data parallel* [370].

Si $i$ es el contador de iteración y $d$ la dimensión del hipercubo, para cada valor de $i$ habrá $2^{d-1}$ intercambios. Después de $d$ iteraciones todos tendrán una carga uniforme dada por $A = (A_1 + ... + A_N)/N$ (promedio de todos los pesos). Todos los procesadores están involucrados en intercambiar datos para cada valor de $i$, es decir, cada procesador intercambia datos con su vecino en la dimensión $i$. El algoritmo general es:

for $i = 1$ to $d$
    enviar $miCarga$ al procesador vecino en la dimension $i$
    recibir $cargaVecino$ desde el procesador vecino en la dimension $i$
    $promedio = (miCarga + cargaVecino)/2$
    $micarga = promedio$
endfor

donde $d$ es la dimensión del hipercubo, $miCarga$ la cantidad de potencial trabajo en el procesador (estimador de cuánto trabajo necesita hacerse), y $cargaVecino$ es similar a $miCarga$ pero para el vecino. La Figura 5.1 muestra la aplicación del método.

Xu y Lau mostraron que un *equal splitting* de carga en una operación entre un par de procesadores podría no llevar a la máxima eficiencia en mesh y toro, aunque puede ser adecuada en el hipercubo [369, 372]. Por esto, puede utilizarse un parámetro de intercambio $\lambda$ para determinar la fracción de exceso de trabajo a migrar entre un par de procesadores, y la eficiencia del método está determinado por $\lambda$. La incorporación de este parámetro dio lugar al método Dimension Exchange Generalizado (GDE) [370, 369, 373].

### 5.4.7 Métodos basados en gradiente

La idea es mantener un contorno de los gradientes formados por las diferencias en las cargas de trabajo. Las cargas en puntos altos del contorno (procesadores altamente cargados) fluirían naturalmente, siguiendo los gradientes, hacia las regiones más bajas (procesadores poco cargados). Dos ejemplos son *Gradient Model* (GM) de Lin y Keller [237, 358] y *Contracting Within a Neighborhood* (CWN) de Shu y Kale [311].

El contorno en GM se denomina *superficie gradiente* y se define en términos de las proximidades de procesadores alta o moderadamente cargados hacia los pocos cargados. Dado que en un entorno distribuido es muy costoso mantener valores de proximidad precisos, usaron una aproximación denominada *superficie de presión* (*pressure surface*). Son un conjunto de presiones propagadas de todos los procesadores y se mantiene

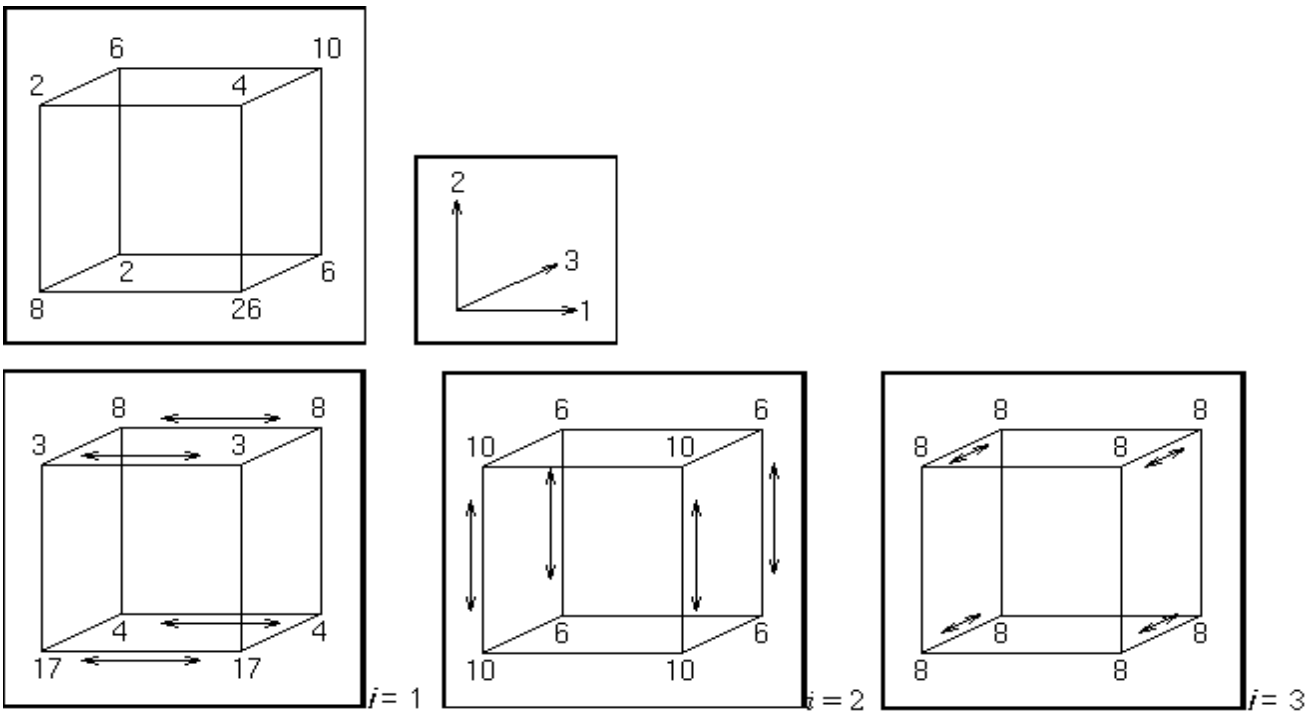


Figura 5.1: Método Dimension Exchange en un hipercubo con $d = 3$

dinámicamente de acuerdo a la distribución de carga. La presión propagada del procesador $i$ ($P_i$) se define como 0 si está poco cargado (respecto de un *threshold*) o $1+min(P_j$ : $j$ *es vecino de* $i$) en otro caso. Basado en la superficie de presión, el trabajo en un procesador sobrecargado migraría hacie el poco cargado más cercano siguiendo el gradiente más pronunciado.

En GM, la migración sólo ocurre entre procesadores muy y poco cargados. Cuando no hay nodos poco cargados en el sistema no se producen migraciones, o dicho en otras palabras, el algoritmo no trataría de transferir carga entre procesadores con carga alta o moderada. Luego, cuando una gran parte de moderadamente cargados súbitamente se convierten en poco cargados se produce una conmoción. El método *Extended Gradient Model* (X-GM) [241] trata de solucionarlo agregando una superficie de succión basada en las proximidades (estimadas) de procesadores no altamente cargados y altamente cargados. Entonces, además de la migración desde muy a poco cargados manejada por la superficie de presión, la superficie de succión causaría migración desde los muy cargados a mínimos locales que pueden ser los moderadamente cargados.

En el CWN cada procesador necesita mantener sólo los índices de carga de trabajo de sus vecinos directos; migra su carga extra (por ejemplo debida a la creación de un proceso) al vecino con menor carga. Un procesador que recibe el proceso lo mantiene para ejecutarlo si es el menos cargado al compararlo con todos sus vecinos; en caso contrario lo forwardea

a su vecino con menos carga. Así, un proceso recién creado viaja a lo largo del gradiente hacia un mínimo local (como el agua fluyendo desde la cima de la montaña hacia la base). Para evitar el *efecto horizonte* (el agua estancada en una superficie plana), el algoritmo impone una distancia mínima que se requiere que viaje un proceso. También fija una distancia máxima para acotar el costo del viaje. Si estos parámetros son ajustables en ejecución se tiene Adaptive Contracting Within a Neighborhood (ACWN).

Todos los métodos basados en gradiente pueden considerarse como una forma de relajación, en la cual la migración de un salto de la carga de trabajo es una aproximación sucesiva hacia un balance global. Los temas a considerar incluyen cuáles procesadores migran trabajo y a quiénes, cuánta distancia puede viajar una carga, y por supuesto la corrección (llegar a un estado balanceado) y eficiencia en implementaciones prácticas.

### 5.4.8 DASUD (*Diffusion Algorithm Searching Unbalanced Domains*)

Es un algoritmo totalmente distribuido que pertenece a la clase *nearest-neighbors*, y está inspirado por la idea de los métodos difusivos iniciados por el emisor (SID) [77, 78, 74, 79, 76]. Durante el proceso de balanceo hay situaciones donde SID no realiza ningún movimiento de carga entre un procesador y sus vecinos mientras el dominio no está balanceado. DASUD incorpora nuevas características para resolver este problema: cuando SID no realiza ningún movimiento en un paso particular, DASUD detecta si el dominio está balanceado o no. Si no lo está, se realizan movimientos de carga adicionales de acuerdo a distintos escenarios. Los movimientos de carga incluyen no sólo la difusión entre un procesador y sus vecinos inmediatos, sino también la capacidad de que un procesador comande la migración de carga de uno de sus vecinos a otro.

Las siguientes acciones resumen los posibles movimientos de carga que realiza cada procesador en presencia de un dominio desbalanceado:

- Acción 1: si el procesador $i$ es el de mayor carga de su dominio, y todos sus vecinos inimediatos tienen la misma carga, entonces $i$ distribuirá su exceso equitativamente entre sus vecinos.

- Acción 2: si $i$ es el de mayor carga de su dominio pero no todos sus vecinos tiene la misma carga, entonces envía una unidad de carga a un procesador de los menos cargados.

- Acción 3: si el dominio de $i$ no está balanceado pero $i$ no es el más cargado, entonces $i$ instruirá a uno de sus vecinos con máxima carga para enviar una unidad de carga a un procesador del vecinaje con carga mínima.

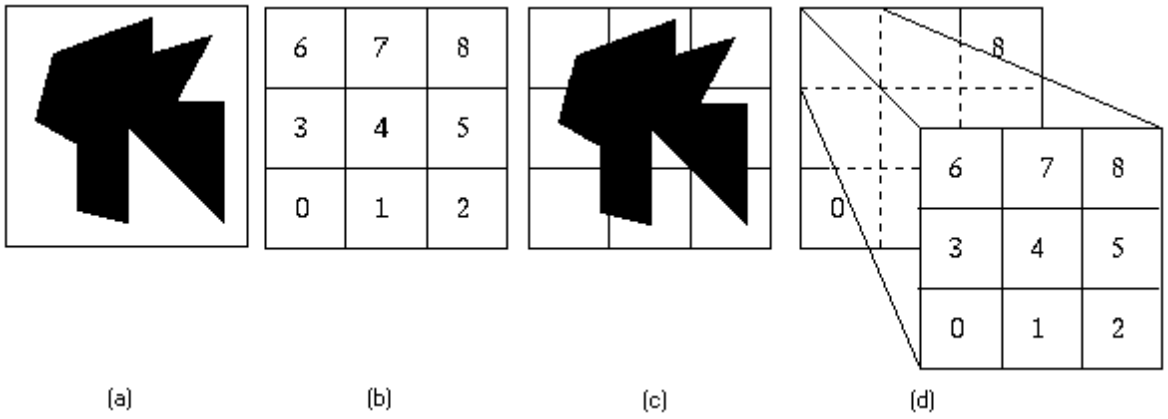


Figura 5.2: Descomposición Scattered. (a) Región. (b) y (c) Descomposición en $N^2$ regiones. (d) Descomposición en gránulos.

Puede encontrarse una descripción de complejidad del comportamiento de DASUD en [75, 78]. Respecto de la estabilidad, DASUD supera a GDE porque logra un grado de balance mayor y la desviación es baja para diferentes topologías y distribuciones respecto de SID y GDE [79].

### 5.4.9 Descomposición scattered

En algunas aplicaciones donde la carga en la red varía fuertemente de un paso al siguiente, usar un balanceador dinámico podría causar que el tiempo total de ejecución sea tan grande como cuando no se usa ninguno (ya que la invocación para balancear se realiza con demasiada frecuencia). La descomposición scattered *no es* es una técnica de balance de carga dinámico, sino que es una técnica de *descomposición de dominio* adecuada para problemas donde la carga puede variar mucho entre procesadores (como en una simulación dinámica o la modelización de una región irregular).

Por ejemplo (Figura 5.2), si se tiene un dominio rectangular con una región irregular, normalmente se subdivide el cuadrado en $N^2$ áreas, se asigna un procesador a cada área ($N^2$ procesadores) y cada uno es responsable de su área. Sin embargo, si la mayoría de la carga de un procesador está determinada por cuánto de la región crítica está contenida en ese procesador, algunos podrían tener mucho más trabajo que otros. La descomposición scattered consiste en descomponer cada uno de los $N^2$ pequeños cuadrados (*templates*) en otro conjunto de $N^2$ cuadrados llamados *gránulos*. Luego, a cada procesador se le asigna un gránulo de cada template. La carga debería estar aproximadamente balanceada.

En general, cada procesador recibe un conjunto disjunto de clusters. Un potencial problema es el overhead de comunicación para los problemas altamente irregulares.

### 5.4.10 Minimización de una función de energía

La idea es representar la carga en la red como una función de energía que puede ser minimizada para encontrar la carga mínima sobre la red. Puede aplicarse tanto como balanceador estático (la función es minimizada antes de la ejecución) o dinámico (se minimiza durante la ejecución).

La mayoría de los problemas a resolver usando una máquina paralela tiene dos partes distintas: cálculo y comunicación (salvo cuando estas partes pueden superponerse). Luego, una manera de maximizar la eficiencia de un sistema es:

1. Minimizar el desbalance de carga por cálculo total (denotado $E_1$), esto es, $E_1 = A_1^2 + .. + A_N^2$, donde $A_i$ es la carga en el procesador $i$ y,
2. Minimizar el total de comunicación en el sistema, es decir $E_2 = \sum_{p=1}^{N} \sum_{q=1,q\neq p}^{N} C_{pq}$, donde $C_{pq}$ es la cantidad de comunicación entre los procesadores $p$ y $q$.

Luego, se minimiza $E = E_1 + E_2$ para balancear la carga en la red. Una técnica que puede usarse para minimizar $E$ es simulated annealing.

#### Balance hidrodinámico

Se trata de un método propuesto para el balance de carga dinámico heterogéneo, basado también en la minimización de una forma de energía. El grafo que representa al sistema se ve como cilindros con líquidos conectados [178]. Cada nodo se asocia con un clindro con líquido que representa la carga de trabajo alocada en él, y hay un canal delgado que une los cilindros si existe una conexión entre los dos procesadores. La solución modeliza el flujo de líquido entre los cilindros. Es intuitivo que el balance global se logra cuando las alturas de las columnas de líquido son iguales. También es obvio que después de lograr el balance global no hay flujo de líquido entre los cilindros y el sistema se encuentra estable.

### 5.4.11 Métodos adaptivos basados en árbol

El método de elementos finitos es usado para la modelización estructural de sistemas físicos. En el mismo, un objeto puede verse como un grafo de elemento finito (grafo conectado y no dirigido que consta de un número de elementos finitos). Cada elemento finito se compone de un número de nodos. El número de nodos está determinado por la aplicación. Debido a las propiedades de intensidad y localidad de computación, resulta atractivo implementar el método sobre multicomputadoras de memoria distribuida [236, 20, 113, 315, 360].

Al paralelizar un programa de aplicación de elementos finitos que usa técnicas iterativas para resolver un sistema de ecuaciones, el programa puede verse como un conjunto de tareas representadas por nodos de un grafo de elementos finitos. Cada nodo representa una cantidad de cómputo y puede ser ejecutado independientemente. En cada iteración, un nodo realiza su computación y necesita obtener datos de los otros antes de la siguiente.

Para ejecutar eficientemente tal aplicación es necesario mapear los nodos del grafo en los procesadores tal que cada uno tenga aproximadamente la misma cantidad de carga y se minimice la comunicación; para tal problema *NP*-completo se propusieron diversas soluciones heurísticas subóptimas [116, 29, 30, 102, 128, 127, 162, 360]. Si el número de nodos no crece durante la ejecución, el algoritmo sólo necesita realizarse una vez. Para una solución adaptiva, el número de nodos crece discretamente debido al refinamiento de algunos elementos en ejecución, lo que puede resultar en desbalance de carga. Debe realizarse un algoritmo de remapeo de nodos o balanceo varias veces para resolver este problema minimizando el costo de comunicación.

En [236], Liao y Chung proponen tres métodos de balance de carga paralelos basados en árbol para tratar eficientemente los problemas de desbalance en aplicaciones de elementos finitos adaptivas sobre máquinas de memoria distribuida: MCSTLB (*maximun cost spanning tree load-balancing*), BTLB (*binary tree load-balancing*) y CBTLB (*condensed binary tree load-balancing*).

Para balancear la carga computacional, MCSTLB primero encuentra un *spanning tree* de costo máximo a partir del grafo de procesadores. Basado en esto, la información de balance de carga global se calcula usando el algoritmo *tree walking* (TWA) [368]. De acuerdo a esta información y a la distribución de carga corriente, se realiza un algoritmo de transferencia. Para BTLB se obtiene un árbol binario a partir del grafo; el cálculo de información global y el método de transferencia son los mismos que para MCSTLB. En CBTLB, los nodos del grafo primero se agrupan en metaprocesadores (cada uno es un hipercubo). Este grafo se denomina *condensed processor graph*, y luego se busca un árbol binario de este grafo; basado en este árbol, se calcula usando un TWA similar. De acuerdo a la información global y la corriente, se realiza un algoritmo de transferencia para balancear la carga de los metaprocesadores y minimizar el costo de comunicación entre ellos. Luego se realiza un método DE para balancear la carga de los procesadores en un metaprocesador. Los resultados experimentales muestran que el tiempo de ejecución de una aplicación usando CBTLB es menor que el de BTLB y el de MCSTLB [236].

### 5.4.12 Métodos estocásticos

Los métodos de balance de carga iterativos estocásticos "arrojan dados" en un intento por llevar al sistema a un estado de equilibrio con alta probabilidad. El más simple es la alocación randomizada ya descripta. Otro enfoque es simulated annealing, que ofrece

más variedad en el control de la aleatoriedad en la redistribución de procesos, haciendo al método menos suceptible de ser "atrapado" en óptimos locales. Enfoques más recientes incluyen los algoritmos genéticos y evolutivos, y las redes neuronales.

### Simulated annealing

Es una potente técnica general para problemas de optimización combinatoria tales como minimización de funciones de varias variables, inspirada en el recocido de cristales en física estadística [202, 267]. La técnica es en esencia Monte Carlo, lo cual simula los movimientos al azar de un conjunto de átomos que vibran en un proceso de calentamiento. Los átomos son más vibrantes a altas temperaturas que a bajas. El objetivo es llegar a una configuración de parámetros de energía baja óptima de los átomos a baja temperatura.

En cada temperatura $T$, se evalúa un conjunto de propuestas aleatorias y permisibles de configuraciones de átomos en términos del cambio de energía $\delta H$. Una configuración de átomos se acepta si representa un decremento de energía; en otro caso, se acepta con una probabilidad condicional de $\exp(-\delta H/T)$, lo que puede ayudar a evitar mínimos locales (*Metropolis Algorithm*). A medida que la temperatura decrece, se acepta cada vez menos incrementos de energía y los átomos eventualmente se establecen en una configuración de baja energía cercana (o idéntica) a la óptima. Cuanto más lento sea el proceso de recocido, pueden explorarse más configuraciones en cada temperatura con mayor probabilidad de llegar a un óptimo. Si bien se encuentran soluciones sub-óptimas para diferentes problemas, el método tiene la desventaja de ser muy lento, por lo que hubo intentos por paralelizarlo.

Cuando se aplica al balance de carga, una configuración corresponde a un estado del sistema (distribución global de carga) y la configuración final corresponde al resultado (un estado balanceado) de la ejecución del procedimiento de balanceo. Bollinger y Midkiff aplicaron simulated annealing a mapeo estático, mejorando métodos determinísticos previos [49]. La aplicación a balance dinámico se realizó por investigadores de Caltech, con una implementación centralizada que contenía un cuello de botella en el balanceador. La paralelización intenta resolverlo distribuyendo la toma de decisión entre una cantidad de procesadores de modo que cada uno pueda ofrecer propuestas parciales. Para esto es necesario tratar el problema de las *colisiones*, que ocurre cuando los cambios propuestos por un procesador conflictúan con los de otro. Una solución es el *rollback*, aunque con gran overhead para mantener la historia del sistema. Otra posibilidad es permitir que ocurran las colisiones, y en cada iteración los vecinos intercambian valores para mantener la consistencia.

Se puede dividir la discusión del algoritmo de simulated annealing paralelo en:

- Descomposición funcional. Explota cualquier paralelismo en la función ($E$), por lo que es adecuado para problemas que requieren grandes tiempos para evaluar $E$

(por ejemplo, si involucra el cálculo del camino mínimo en un grafo). La iteración del algoritmo se mantiene intacta pero la evaluación de $E$ se hace en paralelo. Dado que con frecuencia el cálculo de $E$ contiene sólo paralelismo de grano fino, los overheads de comunicación y sincronización pueden dominar haciendo este enfoque inapropiado para muchas aplicaciones.

- Búsquedas simultáneas independientes. Dados $N$ procesadores, se corre un programa de simulated annealing independiente en cada uno, comenzando a partir de una configuración inicial arbitraria diferente, y usando la misma función $E$ en todos. Cuando terminan todas las ejecuciones, cada procesador pasa su configuración final a un *master* que chequea la mejor. Esto permite la exploración de distintas partes del espacio de búsqueda para lograr una mejor configuración.

- Búsquedas simultáneas interactuantes periódicamente. Similar a la anterior, pero con interacciones entre los procesadores a intervalos regulares de tiempo ($s$, $2s$, ...). Todos usan un schedule de calentamiento y una $E$ común y tienen $N$ configuraciones iniciales distintas. En los intervalos $js < n < (j+1)s$, todos realizan sus pasos independientes usando el mismo schedule de calentamiento ($T_n$). En tiempo $(j+1)s$ cada procesador $P_k$ realiza un paso de recocido a temperatura $T_{(j+1)s}$ para generar una nueva configuración tentativa $Y^k$. La configuración $X^k_{(j+1)s}$ para $P_k$ en tiempo $(j+1)$ está dado por $X^k_{(j+1)s} = Y^1 \diamond Y^2 \diamond \cdots Y^k$, donde $k = 1..n$ ($X = X^i \diamond X^j$ significa $X = X^i$ si $E(X^i) < E(X^j)$ y $X = X^j$ si $E(X^i) > E(X^j)$). Esto requiere la comunicación de $Y^1$ desde $P_1$ a $P_2$, de $Y^2 \diamond Y^1$ de $P_2$ a $P_3$, etc. Al final, $P_N$ contendrá la mejor configuración de toda la corrida.

### Algoritmos genéticos y evolutivos

La computación evolutiva es un área de intenso desarrollo. Esta clase de algoritmos mantiene una población de soluciones posibles que tienen algún proceso de selección basado en el *fitness* de los individuos, y siguen un proceso de *evolución inteligente* para los individuos dado por la aplicación de operadores genéticos (mutación, inversión, selección y *crossover*). Están inspirados en la Teoría de la Evolución de Darwin [170], e imitan los principios de la evolución natural para problemas de optimización de parámetros [129, 250]. Los algoritmos genéticos de John Holland pertenecen a este tipo de métodos basados en evolución.

Los algoritmos genéticos (GA) comienzan con un conjunto de soluciones o individuos (representados por cromosomas) llamado población. Las soluciones en una población se usan para generar una nueva población. Esto se realiza pensando que la nueva población será mejor que su predecesora. En los algoritmos genéticos con poblaciones de tamaño fijo (SGA) los individuos elegidos para crear nuevas generaciones son elegidos de acuerdo a su fitness: aquellos con mayor fitness tendrán mayores chances de reproducción. Esto se

repite hasta que se verifica una condición tal como el número de generaciones o alcanzar un óptimo esperado.

Los algoritmos genéticos con tamaño de población variable (GAVaPS) intentan resolver algunos de los problemas producidos con SGA. No usan ninguna de las variantes de selección para la eliminación de individuos, sino que incorporan el concepto de *edad* del cromosoma (la cual depende de su fitness). Esto es equivalente al número de generaciones que el cromosoma sobrevivió hasta ese momento, y cambia con las generaciones. En consecuencia, la edad del cromosoma reemplaza el concepto de selección. Esto hace que el tamaño de la población no se mantenga constante sino que varía entre generaciones. Además, introducen el concepto de *tiempo de vida* que indica la cantidad de generaciones que un individuo debería sobrevivir (se asigna una sola vez a cada individuo al nacer, y se mantiene constante durante su evolución). Hay distintos métodos para asignar tiempos de vida (proporcional, lineal, bilineal) [250].

La forma genérica de este tipo de algoritmos es la siguiente:

```
Generar los individuos que representan soluciones potenciales
Repetir hasta que el sistema converja
    Evaluar cada individuo
    Elegir el mejor individuo para reproducir
    Reproducir el individuo usando los operadores evolutivos
    Reemplazar los peores viejos individuos por los nuevos
```

Las implementaciones de GA pueden no ser satisfactorias por distintas razones. Por ejemplo, la codificación del problema puede no ser la adecuada y el GA opera en un espacio de búsqueda que no *matchea* con el del problema, o hay un límite en el número de generaciones y el tamaño de la población. Entonces, bajo ciertas condiciones, se produce convergencia prematura hacia óptimos locales. La asignación de tiempos de vida por clases intenta resolver estos problemas [219].

El problema de alocación de tareas a procesadores puede resolverse utilizando soluciones basadas en algoritmos genéticos. Se modeliza la ejecución de un programa paralelo como un grafo, con nodos representando tareas con sus costos de computación y arcos indicando relaciones de precedencia. El enfoque dado en [2] consta de dos fases: en la primera se hace un $k$-particionamiento del grafo minimizando el volumen de comunicación y el costo del desbalance de carga entre los subgrafos (*clusters*), donde $k$ es el número de procesadores. En la segunda fase, los clusters de tareas se asignan a los procesadores (con topología hipercubo) minimizando la distancia de comunicación entre clusters. Otras referencias en el tema de scheduling utilizando algoritmos evolutivos son [250, 118, 288, 376, 263, 103, 104, 105, 298].

### Redes neuronales

Son herramientas cuya característica fundamental es que no tienen un algoritmo predefinido, sino que su comportamiento se va adaptando a medida que *aprenden* en un proceso de entrenamiento. La combinación de elementos (cada uno con su comportamiento) da lugar a la solución del problema. De alguna forma intentan *imitar* el comportamiento del cerebro, y presentan un interesante paralelismo implícito.

Una red neuronal (NN, *neural network*) es un conjunto de elementos de procesamiento simples interconectados. Puede representarse como un grafo dirigido en el cual los nodos representan elementos de procesamiento (neuronas) y las aristas son conexiones entre ellos [163, 114].

Cada neurona puede tener varias conexiones de entrada y genera una única respuesta (que puede replicarse las veces que sea necesario). Para obtener la salida utiliza una función de transferencia basada en las entradas y la memoria local de la neurona. La función, además de brindar la respuesta, puede modificar la memoria local.

Las neuronas pueden agruparse en niveles, y en cada nivel generalmente la función de transferencia es la misma. La encargada de adaptar el comportamiento de entrada/salida es la subfunción denominada *ley de aprendizaje*. Otras formas de aprendizaje comprenden la creación o destrucción de conexiones entre neuronas. Un ejemplo de la aplicación de NN a problemas de asignación de tareas en procesadores puede verse en [381].

## 5.5 Aplicaciones

Los algoritmos de balance de carga se encuentran integrados en diferentes aplicaciones reales, algunas de las cuales se enumeran a continuación. Puede consultarse [289, 39, 125, 34, 231, 192, 53, 27, 299, 193, 217]

- *Branch-and-bound*. Especialmente para *Best-First-Branch-and-Bound* es necesario realizar no sólo balance de carga cuantitativo sino cualitativo para asegurar que todos los procesadores trabajen en buenas soluciones parciales y así reducir el trabajo no esencial (aquel no procesado por un algoritmo *best-first* secuencial).
- *Virtual Data Space* (VDS). Modeliza un área de memoria global en la cual el usuario puede insertar y retirar objetos de datos. Están distribuidos entre los procesadores lo más equitativamente posible con respecto al overhead de balanceo incurrido. Dependiendo del tipo de objeto, VDS ofrece distintas estrategias tales como *load-sharing*, *load-balancing* y *scheduling algorithms* para computaciones multithread. VDS puede utilizarse por ejemplo en la paralelización de aplicaciones *divide-and-conquer* como aislamiento de raíces reales en polinomios.

- Evaluación de árboles de juegos. Los algoritmos de búsqueda en árboles tienen un rol importante en muchas aplicaciones de Inteligencia Artificial. Especialmente en el campo de la búsqueda en árboles de juegos como programas de ajedrez, donde la fortaleza depende en gran parte de la velocidad: cuanto más rápida es la máquina mejor juega. Por lo tanto, hay mucha investigación para acelerar los algoritmos de búsqueda usando máquinas paralelas trabajando en varias transposiciones al mismo tiempo. Luego, la evaluación del árbol en paralelo es otro campo de aplicación de algoritmos de balance de carga dinámicos.

- Cómputo científico. Por ejemplo, los métodos de elementos finitos que normalmente constan de tres fases: generación de la mesh, descomposición de dominio y simulación. En la fase de simulación cada procesador está trabajando en una parte de la mesh; con frecuencia es necesario refinar algunas partes de la mesh lo cual lleva a desbalances de carga.

- *Rendering*. El objetivo de los sistemas realistas de síntesis de imágenes es el *rendering* de alta calidad usando simulación física de la iluminación global. En la versión paralela la imagen se divide en partes que se distribuyen entre los procesadores. Debido a la irregularidad de la escena y la iluminación, los desbalances de carga deben ser ecualizados usando algoritmos dinámicos.

- *Parallel data server*. Consiste de un número de discos y un número de nodos de entrada. Los nodos y los discos se conectan por una red. Para permitir el uso balanceado de los discos, los datos se duplican y almacenan en varios discos, y por lo tanto es necesario usar algoritmos dinámicos para distribuir los pedidos de usuario equitativamente. Un problema mayor es el scheduling de la comunicación resultante entre nodos y discos. Es importante tener en cuenta en este contexto las restricciones de tiempo real para repartir grandes cantidades de datos (video y audio).

## 5.6 Comentarios

El balance estático en general es de menor complejidad que el dinámico, pero también menos versátil y escalable; es posible que presente problemas a la hora de agrandar el problema. Por otra parte, hay casos en que un balance estático difícilmente sea adecuado. Por ejemplo, en un cluster de PCs la carga del procesador puede estar dada por la evolución del programa, pero también por lo que el usuario de la computadora haga. Obviamente el balance estático casi no podría tomar esto en cuenta. Alguna forma de balance dinámico podría detectar este problema en el procesador e intentaría solucionarlo.

De todas formas, no puede establecerse un método efectivo y eficiente en todos los casos. Siempre la elección dependerá de la aplicación y la plataforma de soporte, y en

muchos casos será necesario adaptar o combinar metodos existentes para lograr buena performance.

# Capítulo 6

# Sorting

## 6.1 Introducción

El *sorting* (ordenación) es una de las operaciones más comunes realizadas en una computadora. Muchos algoritmos requieren que los datos se encuentren ordenados para poder accederlos de manera más eficiente [209, 278]. Tiene una importancia adicional para los diseñadores de algoritmos paralelos: con frecuencia es usado para realizar permutaciones de datos en computadoras de memoria distribuida, y estas operaciones pueden utilizarse para resolver problemas en teoría de grafos, geometría computacional o procesamiento de imágenes en tiempo óptimo o casi óptimo.

El sorting es importante dentro del cómputo paralelo por su relación cercana con la tarea de rutear datos entre procesadores, que es parte esencial de algunos algoritmos paralelos. Muchos problemas de *routing* pueden ser resueltos ordenando los paquetes en sus direcciones de destino, mientras varios algoritmos de sorting se basan en esquemas de ruteo para su implementación eficiente [226]. También la operación de ordenación es utilizada con frecuencia en el procesamiento de Bases de Datos, por ejemplo en operaciones con cláusulas *Distinct*, *Order By* y *Group By* en SQL.

La tarea del sorting se define como el acomodamiento de un conjunto desordenado de elementos en orden creciente (o decreciente). Específicamente, para el caso de ordenar en forma creciente, si $a = \langle a_1, a_2, \ldots a_n \rangle$ es una secuencia de $n$ elementos en orden arbitrario, el sorting transforma $a$ en una secuencia creciente $a' = \langle a'_1, a'_2, \ldots a'_n \rangle$ tal que $a'_i \leq a'_j$ para $1 \leq i \leq j \leq n$ y $a'$ es una permutación de $a$.

En el sorting *interno* los datos caben en la memoria principal de la máquina y no se necesita utilizar almacenamiento secundario. Por otro lado, el sorting *externo* involucra ordenar más datos de los que caben en la memoria combinada de todos los procesadores de la máquina paralela, y necesita hacer uso de memoria secundaria.

Los algoritmos de sorting pueden categorizarse como *basados en comparación* y *no basados en comparación* [209]. Los primeros ordenan comparando repetidamente pares de elementos e intercambiándolos si es necesario. Esta operación fundamental se denomina *compare-and-exchange*. Para $n$ items, el sorting secuencial basado en comparación tiene una cota inferior de complejidad en tiempo de $\Theta(n \log n)$. En particular, *mergesort* es $O(n \log n)$, lo que no da lugar a mejoras sustanciales en los algoritmos secuenciales. Pero, a través del paralelismo pueden obtenerse mejoras significativas.

Los algoritmos que no se basan en comparación ordenan usando ciertas propiedades conocidas de los elementos (como por ejemplo su representación binaria o su distribución). La cota inferior de estos algoritmos es $\Theta(n)$. Existen numerosos algoritmos de sorting tanto secuenciales como paralelos. En las secciones siguientes se presentan algunos de los más conocidos.

## 6.2 Algoritmos de sorting secuenciales

### 6.2.1 Selección

Es el algoritmo más sencillo pero también es ineficiente. Para ordenar la secuencia $a = \langle a_1, a_2, \ldots a_n \rangle$ (por ejemplo de menor a mayor) busca el menor y lo intercambia con $a_1$, luego busca el segundo menor a partir de $a_2$ y lo intercambia con éste, y así sucesivamente hasta que todos están ordenados. Esto significa realizar $n - 1$ pasadas. La complejidad es $O(n^2)$.

### 6.2.2 Intercambio o Burbuja (*Bubble Sort*)

Es similar al método de selección en que realiza $n-1$ pasadas para ordenar $n$ items, pero no mueve los elementos grandes distancias sino que a lo sumo realiza correcciones locales de distancia 1: compara ítems adyacentes y los intercambia si están desordenados. Al terminar la primera vuelta se compararon $n-1$ pares y el elemento más grande se arrastró al final. Al terminar la segunda pasada, el segundo máximo estará en la posición $n-1$. La complejidad en tiempo es $\Theta(n^2)$. Puede mejorarse testeando cuándo se producen cambios, y en caso de no haber ninguno, terminar el proceso (Burbuja con centinela).

### 6.2.3 Inserción

Ordena la secuencia de entrada insertando cada elemento $a_i$ entre los $i-1$ anteriores que ya se encuentran ordenados. Para esto, comienza por $a_2$ asumiendo que $a_1$ está ordenado. Si

los dos primeros están desordenados los intercambia. Luego toma $a_3$ y busca su posición correcta con respecto a los dos primeros. En general, para $a_i$, busca su posición con respecto a los $i-1$ anteriores y de ser necesario lo inserta en el lugar adecuado.

### 6.2.4 *Sorting by Merging* (*Mergesort*)

Uilizando la técnica de dividir y conquistar, puede separarse el problema de ordenar una secuencia de $n$ elementos en dividir la secuencia en dos partes de $n/2$ elementos, ordenar cada subsecuencia y mezclar ambas para obtener el resultado final (*merge*). Tiene una complejidad en tiempo de $O(n \log n)$, por lo que es óptimo ya que logra la cota inferior para el problema de sorting que es $\Omega(n \log n)$. Como desventaja, necesita un espacio adicional de $\Theta(n)$ para un arreglo temporario [203].

Este razonamiento puede aplicarse para dividir la secuencia en más partes, y existen versiones iterativas y recursivas del algoritmo de *sorting by merging*. Dado que el proceso de *merge* no es muy costoso, pueden obtenerse reducciones de tiempo de ejecución. Asimismo, es un método fácilmente paralelizable.

### 6.2.5 *Quicksort*

Publicado originalmente por C.A.R Hoare en [167], es uno de los algoritmos secuenciales más rápidos, aplicando recursivamente una estrategia *dividir y conquistar*. En primer lugar, la secuencia a ordenar $a$ se divide en dos partes, tal que todos los elementos de la primera parte $b$ son menores o iguales que todos los elementos de la segunda parte $c$. Luego las dos partes se ordenan separadamente aplicando recursivamente el mismo procedimiento. La recombinación de las partes obtiene la secuencia ordenada.

El primer paso del procedimiento de partición es elegir un elemento de comparación $x$ (*pivot*), de modo que todos los elementos menores que él se ubican en la primera parte y los mayores en la segunda (no interesa de qué lado caen los iguales a $x$).

El comportamiento de mejor caso ocurre cuando en cada paso de recursión el particionamiento produce dos porciones de igual longitud; de ser así, el tiempo de ejecución es $\Theta(n \log n)$ ya que la profundidad de recursión es $\log n$ y en cada nivel hay $n$ elementos a tratar. El peor caso se da cuando en cada paso de recursión se produce un particionamiento desbalanceado (una parte consta de un único elemento y la otra parte del resto), la profundidad de recursión es $n-1$ y Quicksort corre en tiempo $\Theta(n^2)$.

La elección del elemento de comparación $x$ determina qué partición se logra. Si se elige el primer elemento de la secuencia como comparador, se tendría el comportamiento de peor caso cuando la secuencia está inicialmente ordenada. Por lo tanto, es mejor elegir el elemento en el medio de la secuencia como elemento de comparación. Aún mejor

sería tomar el $n/2$-ésimo elemento de la secuencia (la mediana), para obtener la partición óptima. Dado que es posible computar la mediana en tiempo lineal [4], esta variante correría en $O(n \log n)$ aún en el peor caso.

Quicksort es en la práctica el algoritmo de sorting más rápido. Tiene complejidad promedio en tiempo de $\Theta(n \log n)$. Sin embargo, en el (muy raro) peor caso es tan lento como Bubblesort ($\Theta(n^2)$). Hay algoritmos de sorting con una complejidad en tiempo de $O(n \log n)$ aún en el peor caso, pero en promedio son más lentos que Quicksort por un factor constante.

### 6.2.6 *Heapsort*

Publicado originalmente por J.W.J. Williams como *Algorithm 232* en [359], con su complejidad de tiempo $O(n \log n)$ es óptimo. Utiliza una estructura de datos especial llamada *heap*, consistente en un árbol binario donde cada vértice no tiene descendiente directo con un label más grande.

La secuencia a ordenar se almacena como los labels del árbol binario. En la implementación no son necesarias estructuras pointer para representar el árbol, dado que un árbol binario completo puede ser almacenado eficientemente en un arreglo. Si la secuencia a ordenar se organiza como una heap, el elemento más grande puede ser recuperado inmediatamente desde la raíz. Para obtener el siguiente, el resto de los elementos debe ser reorganizado como una heap. El algoritmo puede consultarse en [218].

Un árbol binario completo con $n$ vértices tiene una profundidad de a lo sumo $\log n$. La complejidad en tiempo de heapsort es $T(n) \in O(n \log n)$. El algoritmo es óptimo, dado que se obtiene la cota inferior del problema de sorting.

### 6.2.7 *Shellsort*

Uno de los algoritmos más antiguos, presentado por D.L. Shell en [306]. Es sencillo de implementar aunque su análisis de complejidad es sofisticado. La idea es organizar la secuencia de datos en un arreglo bidimensional y luego ordenar las columnas del arreglo, logrando así una secuencia ordenada parcialmente.

El proceso se repite, pero cada vez con un arreglo con menor cantidad de columnas, hasta que en el último paso el arreglo consta de una sola columna. En cada paso el "orden" de la secuencia se incrementa, hasta que en el último paso está completamente ordenado. El número de operaciones de sorting en cada paso es limitado, debido al preordenamiento obtenido en el paso anterior.

Por ejemplo [218], sea 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 la secuencia a ordenar. En

primer término es organizada en un arreglo con 7 columnas y se ordenan las columnas:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 9 | 0 | 5 | 1 | 6 | ⇒ | 3 | 3 | 2 | 0 | 5 | 1 | 5 |
| 8 | 4 | 2 | 0 | 6 | 1 | 5 | | 7 | 4 | 4 | 0 | 6 | 1 | 6 |
| 7 | 3 | 4 | 9 | 8 | 2 | | | 8 | 7 | 9 | 9 | 8 | 2 | |

Notar que los 8 y 9 están en el final de la secuencia, aunque un elemento chico (2) está allí también. En el paso siguiente, la secuencia se organiza en 3 columnas que nuevamente se ordenan:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 3 | 2 | ⇒ | 0 | 0 | 1 |
| 0 | 5 | 1 | | 1 | 2 | 2 |
| 5 | 7 | 4 | | 3 | 3 | 4 |
| 4 | 0 | 6 | | 4 | 5 | 6 |
| 1 | 6 | 8 | | 5 | 6 | 8 |
| 7 | 9 | 9 | | 7 | 7 | 9 |
| 8 | 2 | | | 8 | 9 | |

La secuencia está casi ordenada. Cuando se la organiza en una columna en el último paso, hay solo un 6, un 8 y un 9 que deben moverse a su posición correcta.

En la implementación, la secuencia puede organizarse como un arreglo unidimensional indexado apropiadamente. Por ejemplo, los elementos en las posiciones 0, 5, 10, 15, etc formarían la primera columna de un arreglo con 5 columnas. Las "columnas" obtenidas indexando de esta manera se ordenan con el método de Inserción, que tiene buena performance con secuencias preordenadas.

El análisis de la complejidad de este método depende en gran parte de la secuencia de organización en columnas que se utilice, y puede encontrarse en [218]. Puede implementarse Shellsort como una *sorting network* si se reemplaza el método de inserción por bubble sort.

### 6.2.8 *Enumeration Sort* o *Counting Sort*

Dada una secuencia de $n$ elementos sobre la cual se definió un orden lineal, el *enumeration sort* computa la posición final de cada elemento en la lista ordenada comparándolo con los otros y contando el número de elementos que tienen menor valor [203]. Si dos o más elementos tienen el mismo valor, el algoritmo debe modificarse levemente tomando en cuenta el índice de cada uno.

Si bien Muller y Preparata en [257] mostraron que un modelo CRCW PRAM no standard puede ejecutarlo en tiempo logarítmico, el algoritmo no es de costo óptimo además de basarse en un modelo PRAM poderoso.

### 6.2.9 *Bucket Sort*

Tiene un tiempo de ejecución promedio menor que la cota inferior de $\Omega(n \log n)$ para el sorting basado en comparación, ya que asume que los $n$ elementos a ordenar están distribuidos uniformemente en el intervalo $[a, b)$. El intervalo se divide en subitervalos de igual tamaño llamados *buckets*, y cada elemento se ubica en el bucket apropiado. Dado que los elementos se distribuyen uniformemente, la cantidad en cada bucket será aproximadamente $n/m$. Luego se ordenan los elementos en cada bucket, obteniendo una secuencia ordenada. El tiempo de ejecución es $\Theta(n \log(n/m))$, y para $m = \Theta(n)$ tiene tiempo lineal $\Theta(n)$.

### 6.2.10 *Sample Sort*

El Bucket Sort requiere que la entrada esté distribuida uniformemente en un intervalo. Sin embargo, en muchos casos la distribución no es tal o es desconocida, y el uso de bucket sort resulta en buckets con cantidades muy distintas de elementos que degradan la performance. En tales situaciones, *Sample Sort* logra mejor performance. La idea es simple: se elige una muestra de tamaño $s$ de la secuencia de $n$ elementos, y el rango de buckets se determina ordenando la muestra y eligiendo $m - 1$ elementos (*splitters*) que dividen la muestra en $m$ buckets de igual tamaño. Luego el algoritmo funciona igual que bucket sort. La performance depende del tamaño de la muestra $s$ y la manera en que es elegida de la secuencia.

### 6.2.11 *Radix Sort*

Es un buen algoritmo inicialmente creado para ordenar números enteros (aunque luego fue extendido a números de punto flotante [343]). Realiza la ordenación sin efectuar realmente comparaciones sobre los datos de entrada, logrando romper la cota inferior teórica de complejidad $O(n \log n)$, aplicable a los métodos basados en comparación. *Radix Sort* es $O(kn)$, con $k = 4$ en la mayoría de los casos; es un método rápido aunque utiliza almacenamiento extra.

Puede decirse que una raíz (*radix*) es una posición en un número. En el sistema decimal, es sólo un dígito en un número decimal (por ejemplo, el 26 tiene dos dígitos, o dos raíces, que son 2 y 6). En hexadecimal la raíz tiene 8 bits (por ejemplo el hexadecimal $0xAB$ tiene dos raíces, $A$ y $B$). El método primero ordena los valores de acuerdo a su primera raíz, luego a la segunda, y así sucesivamente; es un sort multipasada que distribuye cada item en un *bin* o *bucket*, y el número de pasadas es igual al de raíces en los valores de entrada.

Por ejemplo, sean $n$ enteros en el rango $(0, n^2)$ a ser ordenados. En primer lugar,

usando $n$ bins se ubica $a_i$ en el bin $a_i \bmod n$. Luego, se repite el proceso usando $n$ bins y ubicando $a_i$ en al final del bin $\lfloor a_i/n \rfloor$. Esto resulta en una lista ordenada. Si la secuencia de enteros es 36 9 0 25 1 49 64 16 81 4, $n = 10$ y todos los números están en $(0, 99)$, después de la primera fase se tiene:

| Bin | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Contenido | 0 | 1<br>81 | - | - | 64<br>4 | 25 | 36<br>16 | - | - | 9<br>49 |

Notar que en esta fase, cada 'ıtem se ubica en un bin indexado por su dígito decimal menos significativo. Se repite el proceso utilizando el dígito decimal más significativo para ubicar ítems en bins (siempre agregándolos al final). Se tiene:

| Bin | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Contenido | 0<br>1<br>4<br>9 | 16 | 25 | 36 | 49 | - | 64 | - | 81 | - |

Puede aplicarse este proceso a números de cualquier tamaño expresados por cualquier base o raíz.

Otra clase de algoritmos *radix* son los *forward radix sorts*, que examinan primero los bits más significativos y hacen un particionamiento sucesivo yendo hacia los bits menos significativos. Esto tiene la ventaja de que la ordenación puede detenerse cuando se examinaron suficientes bits (para $n$ elementos típicamente está en $\log n$ bits para datos al azar). El problema que presentan es que con frecuencia producen muchos buckets vacíos en las últimas pasadas, provocando alto overhead en el manejo de buckets. Un ejemplo de este tipo de algoritmos es *American flag sort* [248], que intenta evitar los buckets vacíos.

## 6.3 Sorting paralelo

Incluye tanto las versiones paralelas de algoritmos secuenciales clásicos como métodos directamente paralelos. El proceso de paralelizar un algoritmo de sorting secuencial involucra distribuir los elementos en los procesadores disponibles, y esto trae aparejados algunos temas que deben tenerse en cuenta [209].

Uno de estos temas es dónde se almacenan las secuencias de entrada y de salida. En algunos casos pueden estar en un único procesador, o repartidos entre algunos o todos los procesadores, o en una memoria compartida global, o totalmente distribuidos.

Otro punto es cómo se realizan las comparaciones. En el caso de los algoritmos secuenciales, una operación *compare-and-exchange* puede realizarse fácilmente porque los

dos elementos se encuentran en la memoria del procesador. En los paralelos este paso puede no ser tan simple si los elementos no residen en el mismo procesador.

Los problemas de sorting con gran volumen de datos por procesador son los más interesantes y sobre los cuales las máquinas paralelas actuales funcionan mejor, debido a la potencia de cómputo y la capacidad de memoria. Los algoritmos de sorting rápidos generalmente tienen tres componentes: una fase computacional local, una fase intermedia opcional que calcula el destino de las claves para la próxima fase, y una fase de comunicación que mueve las claves entre procesadores y con frecuencia involucra una transformación del conjunto de datos completo.

Un desafío fundamental para el cómputo paralelo es obtener algoritmos de alto nivel, independientes de la arquitectura, que ejecuten eficientemente en máquinas paralelas de propósito general. Este desafío también existe para los problemas de sorting paralelo, además de los otros tradeoffs clásicos tales como balance de carga, overhead, comunicación y localidad que deben considerarse. La distribución de probabilidad del conjunto de datos puede tener un impacto significativo sobre la performance de ciertos algoritmos de sorting, ya que puede influir en la manera en que se balancea la carga, la cantidad de comunicación necesaria y la localidad de los datos.

### 6.3.1 *Sorting networks*

En la búsqueda de métodos de ordenación rápidos se diseñaron una cantidad de redes que ordenan $n$ elementos en tiempo significativamente menor que $\Theta(n \log n)$. Se basan en un modelo de red de comparación (*sorting network*), en el cual se realizan simultáneamente varias operaciones de comparación [209, 203].

El componente clave de estas redes es un *comparador*, dispositivo con dos entradas $x$ e $y$ y dos salidas $x'$ e $y'$. En un *comparador creciente*, $x' = min\{x, y\}$ e $y' = max\{x, y\}$; para un *comparador decreciente*, $x' = max\{x, y\}$ e $y' = min\{x, y\}$. Usualmente una sorting network se compone de una serie de columnas que contienen un número de comparadores conectados en paralelo, y cada columna realiza una permutación.

La *profundidad* de una red es el número de columnas que contiene, e indica la cantidad de pasos paralelos que implica la ordenación, y la *longitud* es el número de comparaciones-intercambios usados ( [175] muestra cotas superiores e inferiores para estos valores). Una de las sorting networks más estudiadas ha sido la de 16 entradas [203, 379]. Puede convertirse cualquier sorting network en un algoritmo secuencial emulando los comparadores por software y realizando las comparaciones de cada columna secuencialmente. El comparador es emulado por una operación *compare-and-exchange*.

### 6.3.2 *Bitonic Sort*

Una red de sorting bitonica ordena $n$ elementos en tiempo $\Theta(\log^2 n)$. La operación clave es el reacomodamiento de una secuencia bitónica en una ordenada. Una *secuencia bitónica* es una secuencia de elementos $\langle a_0, a_1, \ldots a_{n-1}\rangle$ con la propiedad de que (1) existe un índice $i$, $0 \leq i \leq n-1$, tal que $\langle a_0, \ldots, a_i\rangle$ es monótonamente creciente y $\langle a_{i+1}, \ldots, a_{n-1}\rangle$ es monótonamente decreciente, o (2) existe un shift cíclico de índices tal que se satisface (1).

Por ejemplo, $\langle 1, 2, 4, 7, 6, 0\rangle$ es bitónica porque primero crece y luego decrece; de manera similar, $\langle 8, 9, 2, 1, 0, 4\rangle$ también es bitónica, ya que es un shift cíclico de $\langle 0, 4, 8, 9, 2, 1\rangle$. Gráficamente, una secuencia bitónica contiene a lo sumo un "pico" y un "valle".

Un paso *compare-exchange* puede particionar una secuencia bitónica en dos bitónicas. Si $n$ es par, $n/2$ comparadores son suficientes para transformar una bitónica de $n$ valores $\langle a_0, a_1, \ldots a_{n-1}\rangle$ en dos de $n/2$ valores, $\langle min(a_0, a_{n/2}), min(a_1, a_{n/2+1}), \ldots min(a_{n/2+1}, a_{n-1})\rangle$ y $\langle max(a_0, a_{n/2}), max(a_1, a_{n/2+1}), \ldots max(a_{n/2+1}, a_{n-1})\rangle$ tal que ningún valor de la primera secuencia es mayor que ninguno de la segunda. Aplicando este paso recursivamente se obtiene una secuencia ordenada. En otras palabras, dada una secuencia bitónica de longitud $n = 2^k$, con $k > 0$, entonces $k$ pasos compare-exchage son suficientes para obtener una secuencia ordenada [32, 278].

*Bitonic Sort* toma una secuencia bitónica y la transforma en una lista ordenada que puede pensarse como la mitad de una secuencia bitónica de dos veces la longitud. Si una secuencia bitónica de longitud $2^m$ se ordena ascendentemente, mientras una secuencia adyacente de longitud $2^m$ se ordena en forma descendente, entonces luego de $m$ pasos compare-exchange la secuencia combinada de longitud $2^{m+1}$ es bitónica. Una lista de $n$ elementos a ordenar puede verse como un conjunto de $n$ secuencias desordenadas de longitud 1 o como $n/2$ secuencias bitónicas de longitud 2. Por lo tanto se puede ordenar cualquier secuencia de elementos mezclando sucesivamente secuencias bitónicas más y más grandes. Dados $n = 2^k$ elementos desordenados, una red con $k(k+1)/2$ niveles es suficiente, y cada nivel contiene $n/2 = 2^{k-1}$ comparadores. Luego, el número total de comparadores es $2^{k-2}k(k+1)$. La ejecución paralela en cada nivel toma tiempo constante, y $k(k+1)/2 = \log n(\log n + 1)/2$. Entonces, el algoritmo tiene complejidad en tiempo $\Theta(log^2 n)$ [32, 278].

El algoritmo de sorting bitónico puede mapearse a diferentes arquitecturas (*shuffle-exchange*, mesh bidimensional, hipercubo) [278, 209].

### 6.3.3 *Odd-Even transposition sort*

Variante de Bubble Sort, diseñado para el modelo de arreglo de procesadores en el que los elementos de procesamiento se organizan en una mesh unidimensional o un anillo. Ordena $n$ elementos en $n$ fases ($n$ par), cada una de las cuales requiere $n/2$ operaciones

compare-exchange. Alterna entre las fases impar y par.

Sea $\langle a_1, \ldots, a_n \rangle$ la secuencia a ordenar. Durante la fase par, los elementos con índice par son comparados con sus vecinos derechos, y si están fuera de secuencia se intercambian; de manera similar, durante la fase impar, los elementos con índices impar son comparados con sus vecinos derechos, e intercambiados si es necesario. Después de $n$ fases de intercambios par-impar la secuencia está ordenada. Cada fase requiere $\Theta(n)$ comparaciones, y hay un total de $n$ fases, por lo que la complejidad secuencial es $\Theta(n^2)$.

Es sencillo paralelizar el algoritmo. Durante cada fase, las operaciones *compare-exchange* pueden realizarse simultáneamente. En el caso de un elemento por procesador ($n$ procesadores), y topología anillo, el elemento $a_i$ reside inicialmente en el procesador $P_i$. Durante la fase impar, cada procesador con label impar hace *compare-exchange* de su elemento con el que reside en su vecino derecho, y durante la fase par, cada procesador con label par hace lo mismo con su vecino derecho. Cada fase requiere tiempo $\Theta(1)$, y como hay $n$ fases, el tiempo paralelo de esta formulación es $\Theta(n)$. Dado que la complejidad secuencial del mejor algoritmo de sorting para $n$ elementos es $\Theta(n \log n)$ esta formulación no es de costo óptimo ya que el producto procesador-tiempo es $\Theta(n^2)$.

Para obtener una formulación paralela de costo óptimo se usan menos procesadores ($p < n$), y a cada uno se le asigna un bloque de $n/p$ elementos que ordena internamente (usando *mergesort* o *quicksort*) en tiempo $\Theta((n/p)\log(n/p))$. Luego, los procesadores ejecutan $p$ fases de operaciones *compare-split* (quedándose con el bloque que le corresponde luego de comparar), y al final la lista está ordenada. El tiempo paralelo es $T_p = \Theta((n/p)\log(n/p)) + \Theta(n) + \Theta(n)$ (sorting local + comparaciones + comunicaciones). El odd-even transposition sort es de costo óptimo cuando $p = O(\log n)$.

### 6.3.4 *Quicksort* paralelo

Quicksort puede ser paralelizado de diferentes maneras. En primer lugar, sea una solución *ingenua*. Al particionar la secuencia en dos, la llamada para ordenar cada parte implica dos subproblemas completamente independientes que pueden resolverse en paralelo. Entonces, una forma de paralelizar quicksort es ejecutarlo inicialmente en un solo procesador; luego, cuando el algoritmo realiza sus llamados recursivos, se asigna uno de los subproblemas a otro procesador. Cada uno de estos procesadores ordena su arreglo usando quicksort y asigna uno de sus subproblemas a otro. El algoritmo termina cuando las secuencias ya no pueden ser particionadas, y en ese caso cada PE mantiene un elemento y la secuencia ordenada puede recuperarse atravesando los procesadores.

Esta formulación usa $n$ procesadores para ordenar $n$ elementos. La principal desventaja es que el particionamiento de la secuencia en dos lo hace un único procesador secuencialmente. Dado que un procesador debe particionar la secuencia original el tiempo de ejecución está acotado inferiormente por $\Omega(n)$, y la formulación no es de costo óptimo

pues el producto procesador-tiempo es $\Omega(n^2)$.

Otro quicksort paralelo es el siguiente. Una cantidad de procesos idénticos, uno por procesador, ejecutan el algoritmo paralelo. Los elementos a ordenar se almacenan en un arreglo en memoria global. Una pila en esa memoria almacena los índices de los subarreglos que todavía están desordenados. Cuando un proceso no tiene trabajo, intenta desapilar los índices de un subarreglo desordenado; si tiene éxito, el procesador particiona el subarreglo en dos más chicos conteniendo los elementos menores o iguales y mayores que el *pivot* respectivamente; finalmente, apila los índices de un subarreglo en la pila global y repite el proceso de particionamiento para el otro subarreglo.

Para obtener un quicksort eficiente es fundamental realizar el particionamiento en paralelo. Para ver por qué, sea la ecuación de recurrencia $T(n) = 2T(n/2) + \Theta(n)$ que da la complejidad del quicksort para la elección óptima del pivot. El término $\Theta(n)$ se debe al particionamiento. Si se compara con la complejidad global del algoritmo ($\Theta(n \log n)$), se puede pensar al quicksort consistente en $\Theta(\log n)$ pasos cada uno de tiempo $\Theta(n)$ para partir la secuencia. Luego, si el paso de particionamiento se realiza en tiempo $\Theta(1)$ usando $\Theta(n)$ procesadores, es posible obtener un tiempo paralelo de $\Theta(\log n)$ que lleva a una formulación de costo óptimo. Sin embargo, sin paralelizar el particionamiento, lo mejor que puede hacerse manteniendo la optimalidad de costo es usar sólo $\Theta(\log n)$ procesadores para ordenar $n$ elementos en tiempo $\Theta(n)$.

Es difícil particionar una secuencia de tamaño $n$ en dos más chicas en tiempo $\Theta(1)$ usando $\Theta(n)$ procesadores para la mayoría de los modelos de computación paralela. Los algoritmos conocidos son para los modelos PRAM abstractos. A causa del overhead de comunicación, este paso toma más tiempo que $\Theta(1)$ en mesh e hipercubos. En [209] se presentan formulaciones de quicksort para un CRCW PRAM (tiempo $\Theta(\log n)$ en un PRAM $n$-procesador), un hipercubo (tiempo $\Theta((n/p)\log(n/p))+\Theta((n/p)\log p)+\Theta(\log^2 p)$ con $(n/p)$ elementos por procesador y selección óptima del pivot) y una mesh ($O(\sqrt{n}\log n)$ con $n$ procesadores). La formulación PRAM es significativamente más escalable que las otras dos, y la única que puede usar $n$ procesadores para ordenar $n$ elementos en tiempo $\Theta(\log n)$. En el hipercubo, depende fuertemente de la calidad de los pivots, logrando buena escalabilidad si la selección de los pivots es óptima. En la mesh, la formulación del quicksort tiene una función de isoeficiencia exponencial y es práctica sólo para valores chicos de $p$.

### Hiperquicksort

Dado que el speedup obtenible en el quicksort paralelo está restringido por el tiempo que toma realizar el particionamiento inicial, se desarrollaron diversos algoritmos para poner a todos los procesadores a trabajar inmediatamente. El hiperquicksort es uno de ellos, adecuado para implementar en un hipercubo [278].

Dada una lista de valores inicialmente distribuidos equitativamente entre los procesadores, se define que la lista está ordenada cuando (1) la lista de valores de cada procesador está ordenada, y (2) el valor del último elemento en la lista de $P_i$ es menor o igual que el valor del primer elemento en la lista de $P_{i+1}$, para $i$ entre 0 y $p-2$. Los valores ordenados no necesitan estar distribuidos entre los procesadores. Para desarrollar un algoritmo eficiente se aplica la estrategia de dejar a cada procesador resolver un subproblema usando el algoritmo secuencial más eficiente, y luego usar un algoritmo paralelo eficiente en comunicación para generar la solución final.

En la primera fase de hiperquicksort cada procesador usa quicksort para ordenar su lista local de valores. En este punto, cada procesador tiene una lista ordenada de valores que satisfacen (1), pero no (2). Hiperquicksort es un algoritmo recursivo que usa un enfoque *divide-and-conquer* para cumplir con la segunda condición. Durante cada paso de la segunda fase del algoritmo, un hipercubo se divide en dos subcubos. Cada procesador envía valores a su *partner* en el otro subcubo, luego cada procesador mezcla los valores que tiene con los valores que recibe. El efecto de esta operación *split-and-merge* es dividir un hipercubo de valores ordenados en dos hipercubos de modo que cada procesador tenga una lista ordenada de valores, y el valor más grande en el hipercubo menor es menor que el valor más chico en el hipercubo superior. Después de $d$ pasos *split-and-merge*, el hipercubo original $2^d$ procesador fue dividido en $2^d$ hipercubos uniprocesador, y la condición (2) se satisface.

### 6.3.5 *Bucket sort* paralelo

La paralelización es directa. Sean $n$ los elementos a ordenar y $p$ los procesadores. Inicialmente, a cada procesador se le asigna un bloque de $n/p$ elementos, y el número de buckets se elige $m = p$. El algoritmo consta de tres pasos. En el primero, cada procesador particiona su bloque en $p$ subbloques (uno por cada bucket). En el segundo paso, cada procesador envía subbloques a los procesadores adecuados. Luego de esto, cada uno tiene sólo los elementos que pertenecen al bucket que le asignaron. En el último paso, cada procesador ordena su bucket internamente usando un algoritmo secuencial óptimo.

La complejidad en tiempo del bucket sort sobre un hipercubo con ruteo *cut through* es $\Theta(n/p)+\Theta(n/p)+\Theta(p\log p)+\Theta(n/p)$ (el primer término corresponde al particionamiento en bloques, los dos siguientes al broadcast *all-to-all* personalizado, y el último al sort local [209]). La performance es mejor que muchos de los otros algoritmos, aunque siempre en el caso de que los elementos se encuentren distribuidos uniformemente en un intervalo conocido.

### 6.3.6 *Sample sort* paralelo

Sea $p$ el número de procesadores, y como en bucket sort, sea $m = p$ (cantidad de buckets). A cada procesador se le asigna un bloque de $n/p$ elementos que ordena secuencialmente. Después elige $p-1$ elementos equidistantes en el bloque ordenado. Cada procesador envía su muestra a uno de ellos ($P_0$) que ordena los $p(p-1)$ elementos de muestra y elige los $p-1$ *splitters*. Finalmente $P_0$ hace broadcast de los splitters a todos los procesadores, y el algoritmo funciona igual que bucket sort.

Si se asume que los elementos almacenados en cada procesador están distribuidos uniformemente, cada subbloque tiene $\Theta(n/p^2)$. La complejidad en tiempo sobre un hipercubo con ruteo *cut through* es $\Theta((n/p)\log(n/p)) + \Theta(p^2 \log p) + \Theta(p \log(n/p)) + \Theta(n/p) + O(p \log p)$ (el primer término corresponde al sort local, el segundo al sorting de las muestras, el siguiente al particionamiento de bloque y los dos últimos a la comunicación [209]).

En [159] se presenta una modificación del sample sort que usa sólo dos rondas de comunicación personalizada *all-to-all* en un esquema que logra buen balance de carga virtualmente sin overhead.

### 6.3.7 *Parallel Sorting by Regular Sampling* (PSRS)

Desarrollado por Li *et al.* en [233], mostró ser efectivo para una gran variedad de arquitecturas MIMD. Tiene buenas propiedades de balance de carga, necesidades de comunicación modestas y buena localidad de referencia de memoria. Si no hay claves duplicadas, garantiza balancear el trabajo entre los procesadores dentro de un factor de dos del óptimo en teoría, sin importar la distribución de valores de datos, y dentro de un pequeño porcentaje del óptimo en la práctica. No es necesariamente el mejor algoritmo de sorting paralelo para alguna máquina específica, pero obtiene buena performance en un amplio espectro de máquinas.

PSRS es una combinación de un sort secuencial, una fase de balance de carga, un intercambio de datos y un merge paralelo. Aunque puede usarse cualquier algoritmo de sorting y merge secuencial, PSRS se demostró usando quicksort y merge 2-way sucesivo. Dados $n$ items de datos (con índices 1, 2, 3, ..., $n$) y $p$ procesadores (1, 2, 3, ..., $p$), PSRS consta de 4 fases (esta descripción difiere levemente de la de [307]). La Figura 6.1 muestra la aplicación del método con $n = 36$, $p = 3$ y claves 0, 1, ..., 35. Por brevedad, sea $\rho = \lfloor p/2 \rfloor$ y $w = n/p^2$.

1. *Fase Uno. Ordenar datos locales.* A cada procesador se le asigna un bloque contiguo de $n/p$ items. Los bloques asignados a distintos procesadores son disjuntos. Cada procesador, en paralelo, ordena su bloque usando quicksort secuencial.

   Comienza la heurística de balance de carga *regular sampling*. Los $p$ procesadores,

Figura 6.1: Parallell Sorting by Regular Sampling (PSRS)

en paralelo, eligen los items en los índices locales 1, $w+1$, $2w+1$, ..., $(p-1)w+1$ para formar una muestra representativa del bloque ordenado localmente. Los $p^2$ items seleccionados, $p$ de cada uno de los $p$ procesadores, son una *regular sample (RS)* del arreglo completo. Las RS locales representan las claves y su distribución de valor en cada procesador.

En la Figura 6.1, a cada procesador se le asignan $n/p = 12$ claves contiguas para ordenar. Cada uno toma tres muestras, en los índices 1, 5 y 9 pues $w = n/p^2 = 4$, para formar la RS local. Notar que la distancia entre los índices de las muestras es de tamaño fijo.

2. *Fase Dos. Encontrar pivots y particionar.* Un procesador designado reúne y ordena las RS locales. Se eligen $p-1$ pivots a partir de la RS ordenada, en los índices $p+\rho$, $2p+\rho$, $3p+\rho$, ..., $(p-1)+\rho$. Cada procesador recibe una copia de los pivots y forma $p$ particiones a partir de sus bloques locales ordenados. Una partición es contigua internamente y es disjunta de las otras particiones.

   En la Figura 6.1, las 9 muestras son reunidas y ordenadas. A partir de esta lista $(0, 3, 7, 10, 13, 16, 22, 23, 27)$, se eligen $p-1 = 2$ pivots. Los pivots son 10 y 22 (en los índices 4 y 7), dado que $\rho = \lfloor p/2 \rfloor = 1$. Todos los procesadores crean entonces 3 particiones.

3. *Fase Tres. Intercambio de particiones.* En paralelo, cada procesador $i$ mantiene para sí la $i$-ésima partición y asigna la $j$-ésima partición al $j$-ésimo procesador. Por ejemplo, el procesador 1 recibe la partición 1 de todos los procesadores. Por lo tanto, cada procesador mantiene una partición y reasigna $p-1$ particiones.

   Por ejemplo, en la Figura 6.1, el procesador 3 envía la lista $(3, 4, 5, 6, 10)$ al procesador 1, envía $(14, 15, 20, 22)$ al procesador 2, y mantiene $(26, 31, 32)$ para él.

4. *Fase Cuatro. Merge de particiones.* Cada procesador en paralelo mezcla sus $p$ particiones en una única lista que es disjunta de las de los otros procesadores. La concatenación de todas las listas es la secuencia ordenada final.

   Notar que en la Figura 6.1, las claves en las particiones mezcladas finales en cada procesador son también particionadas por los pivots 10 y 22. La lista ordenada final se distribuye a los tres procesadores.

En una máquina MIMD con memoria distribuida es necesario comunicar la información con los siguientes mensajes: las RS locales de la Fase Uno ($p$ mensajes de tamaño $O(p)$), los pivots de la Fase Dos ($p$ mensajes de tamaño $O(p)$) y las particiones de la Fase Tres ($p$ procesadores enviando $p-1$ mensajes de tamaño $O(n/p)$). En una arquitectura de memoria compartida, toda la información se comunica a través de la misma. En particular, la Fase Tres se reduce a leer y escribir particiones desde y hacia memoria.

Intuitivamente la noción de un RS para estimar la distribución de valores de las claves es atractiva. Muestreando los bloques ordenados localmente de todos los procesadores, y

no sólo un subconjunto, todo el arreglo de datos está representado. Al muestrear después que los bloques locales fueron ordenados, se captura la información de orden. Dado que los pivots, como se eligen en la Fase Dos, dividen la RS en particiones casi iguales, deberían dividir también el arreglo de datos entero en particiones casi iguales. También, los intervalos de distancia fija de la heurística de *regular sampling* permiten un análisis formal de su efectividad.

La descripción de PSRS dada especifica que el número de muestras, $s$, tomadas por cada procesador en la Fase Uno es igual a $p$ (es decir, $s = p$). Sin embargo, es una extensión natural y común de los algoritmos basados en sampling considerar las técnicas de *undersampling* ($s < p$) y *oversampling* ($s > p$). Intuitivamente, dado que el número de muestras representa la cantidad de información disponible para la heurística de balance, el undersampling resulta en un balance más pobre y el oversampling en uno mejor.

PSRS combina varios de los aspectos exitosos de los algoritmos de sorting MIMD que lo preceden, e introduce la noción simple pero efectiva de una RS para ayudar a elegir buenos pivots para el merge paralelo final. La RS es la diferencia clave, no cosmética, entre PSRS y algoritmos de sorting similares.

La complejidad computacional general de PSRS es $O((n/p)\log n + p^2 \log p + (n/p)\log p$. Si $n \geq p^3$, el primer término domina y la complejidad es $O((n/p)\log n)$, que es de costo óptimo.

Respecto del balance de carga, en la Fase Uno, la carga está distribuida equitativamente en todos los procesadores. La Fase Dos es en gran parte secuencial. La Fase Tres depende de las propiedades de comunicación de la máquina. Sobre el balance de carga en la Fase Cuatro, en [233] se presenta una cota inferior y una cota superior ajustada sobre el número de items que cada procesador debe mezclar. El análisis es una función del número de items a ser ordenados, el número de procesadores, el tamaño de muestra y el número de veces que un ítem de datos está duplicado.

### 6.3.8 *Parallel Merge Sort*

El *Parallel Merge Sort* usando el modelo PRAM fue reportado con un mejor tiempo de ejecución de $O(\log n)$ para ordenar $n$ claves usando $n$ procesadores [70]. Consta de dos fases: una fase local de ordenación y una de merge. La primera produce claves ordenadas en cada procesador. Luego en la fase de merging, los procesadores las mezclan en $\log p$ pasos; en cada paso se mezclan dos subsecuencias, logrando así porciones ordenadas mayores. Un problema es que la mitad de los procesadores van quedando ociosos en cada uno de los $\log p$ pasos, degradando la eficiencia [191].

Una variante posible es reducir los pasos de merging haciendo que un procesador mezcle más de dos subsecuencias, hasta el caso en el cual un único procesador realiza el

merge en un solo paso. Esto reduce el costo de comunicación, un punto importante en el caso de que la plataforma de soporte sea una máquina con memoria distribuida tipo NOW. Por el lado de las desventajas, si el número de procesadores utilizado crece, el encargado del merge puede convertirse en cuello de botella.

En [191], Jeon presenta el algoritmo Load-Balanced Parallel Merge Sort, que redistribuye cada una de las listas parcialmente ordenadas de modo que cada procesador almacene un número aproximadamente igual de claves, y todos tomen parte del proceso de merge durante la ejecución.

### 6.3.9 Comentarios

Pueden encontrarse en la literatura numerosos algoritmos de sorting paralelo, y está fuera del objetivo de esta Tesis la descripción de todos ellos. A modo de referencia pueden citarse *parallel merge-all sort*, *parallel binary merge sort*, LBOS (*load balancing optimization sort*), LBOS-F (*load balancing optimization sort fragmet feature*), *parallel partitioned sort*, *partitioned with redistribution sort* (todos ellos para ordenación en sistemas de Bases de Datos multiprocesador), *flashsort*, *parallel radix sort*, *partitioned parallel radix sort*, *shearsort*, *gridsort*, *periodic balanced sort*, *columnsort*, *padded sort*, *heterosort*, *LS3-Sort*, 4-*way Mergesort*, *rotatesort*, 3n-*Sort*, etc. Asimismo, se han realizado abundantes estudios comparativos de métodos sobre diferentes arquitecturas [242, 377, 134, 13, 226, 352, 213, 295, 24, 43, 122, 224, 243, 218].

Dado el objetivo de esta Tesis, en lo que sigue se tratará en particular el caso de una variante de *parallel merge sort* o *sorting by merging* paralelo sobre un cluster de PCs.

# Capítulo 7

# Sorting Paralelo con Balance de Carga Dinámico

## 7.1 Introducción

En el Capítulo 5 se trató el tema del balance de carga en sistemas paralelos. Uno de los objetivos que se pretende lograr es una mejora (disminución) del tiempo de ejecución mediante una optimización del trabajo total a realizar.

Puede observarse que gran parte de los métodos (tanto estáticos como dinámicos) son aplicables a problemas en los que existe alguna forma de conocer cuál es la carga. Por ejemplo, en simulaciones utilizando grillas donde la carga está relacionada con la cantidad de puntos a procesar, o donde la resolución de la grilla puede cambiar.

En otros casos, el balance se realiza teniendo en cuenta la expresión de la carga en función de la cantidad de tareas. Esto no es suficiente en los casos en que las tareas no tienen todas la misma carga computacional, sino que son dependientes de los datos. En particular, es complejo encontrar cómo afecta la distribución de los datos sobre la carga de trabajo y sobre la performance.

Resulta interesante (y complejo) atacar el problema del balance de la carga de trabajo en aplicaciones con comportamiento dinámico y donde no se conoce a priori el trabajo total a realizarse, ya que depende no sólo del tamaño de los datos de entrada sino de determinada característica de los mismos. Gran parte de los algoritmos de sorting, cuya importancia se analizó en el Capítulo 6, se encuentran dentro de esta clase.

Algunas técnicas, como por ejemplo Parallel Sorting by Regular Sampling [233] intentan balancear la carga mediante un muestreo inicial de los datos a ordenar y una distribución de los mismos de acuerdo a los *pivots*. Otras, como el Load-Balanced Pa-

rallel Merge Sort [191], redistribuyen listas parcialmente ordenadas de modo que cada procesador almacene un número aproximadamente igual de claves, y todos tomen parte del proceso de merge durante la ejecución.

Esta Tesis presenta un nuevo método que balancea dinámicamente la carga basado en un enfoque diferente, buscando realizar una redistribución del trabajo utilizando un *estimador* que permita *predecir* la carga de trabajo pendiente. La noción principal es la de distribuir inicialmente entre las tareas un porcentaje del trabajo total a realizar, dejando una parte reservada; luego de un período de tiempo, conociendo lo realizado por cada tarea, estimar el trabajo restante en cada una y distribuir la porción reservada entre ellas.

Interesa observar cómo se comporta el mecanismo en la realidad y no sólo respecto de los órdenes de magnitud, ya que en muchos casos las constantes ocultas no permiten tener una idea definitiva de los resultados.

El método propuesto es aplicado en particular al problema de ordenación como una variante de *Sorting by Merging* Paralelo, esto es, una técnica de sorting interno y basado en comparación. Las ordenaciones en los bloques se realizan mediante el método de Burbuja o *Bubble Sort* con centinela.

Se realizaron pruebas con secuencias de datos en las cuales las claves aparecen una única vez y el valor máximo de clave coincide con la cantidad de datos (*secuencias de datos completas*). Esto permite conocer estáticamente, mediante un cálculo basado en las posiciones iniciales de las claves, la cantidad de trabajo a realizar (en términos de comparaciones e intercambios). Esta cantidad de trabajo se encuentra afectada por el *grado de desorden* que presentan los datos. Si bien esta situación en cuanto a los datos no se presenta como demasiado realista, permitió estudiar la evolucion de la cantidad de trabajo en cada "vuelta" del algoritmo, de modo de establecer una ley de estimación de trabajo futuro.

Luego se experimentó con secuencias de datos al azar. En este caso, el trabajo no puede conocerse estáticamente aunque como en el caso anterior también se observó que disminuye en cada vuelta. Esto se utilizó para obtener una estimación del trabajo restante esperado a partir de una iteración determinada, y basarse en el mismo para corregir la distribución de la carga.

Con esta idea, el método propuesto no distribuye inicialmente entre las tareas todos los datos a ordenar, sino que se reserva un porcentaje de los mismos. Luego de una determinada cantidad de "vueltas" (en particular, con el 5% de las iteraciones), estima el trabajo restante de cada tarea basado en lo ya realizado, y distribuye dinámicamente la porción reservada de manera inversamente proporcional al mismo.

El esquema planteado utiliza el paradigma *master/slave*, y posee buenas propiedades ya que requiere poca comunicación sobre una arquitectura MIMD con comunicación por

bus como un cluster de PCs. También puede ejecutarse en máquinas de memoria compartida. Las características principales del método son su sencilllez, efectividad, comunicación limitada, y posibilidad de aplicación a diferentes problemas. Además, en el caso particular del sorting, podría realizarse un análisis similar con otro algoritmo para la ordenación de las partes (el punto crítico es lograr un buen estimador del trabajo).

El soporte utilizado para la experimentación fue lenguaje C y librería MPI sobre un cluster de PCs homogéneas (la descripción de la arquitectura se encuentra en el Apéndice B). En el Capítulo 8 se presentan resultados que muestran la bondad de las estimaciones y el método balancea la carga de trabajo adecuadamente en un alto porcentaje de los casos.

## 7.2 Secuencias de datos completas. Medición del desorden

Como se expresó en el Capítulo 6, el sorting puede definirse como el acomodamiento de un conjunto desordenado de elementos en orden creciente (o decreciente). De aquí en más, se supondrá que el resultado del sorting será una secuencia creciente. Todos los resultados son válidos para el caso decreciente.

Específicamente, si $a = \langle a_1, a_2, \ldots a_n \rangle$ es una secuencia de $n$ elementos en orden arbitrario, el sorting transforma $a$ en una secuencia creciente $a' = \langle a'_1, a'_2, \ldots a'_n \rangle$ tal que $a'_i \leq a'_j$ para $1 \leq i \leq j \leq n$ y $a'$ es una permutación de $a$.

### 7.2.1 Algoritmos básicos

El primero de los algoritmos básicos utilizados es el *bubble sort con centinela*(BSCen). El mismo realiza *a lo sumo* $n-1$ pasadas o vueltas para ordenar $n$ items. Compara ítems adyacentes y los intercambia si están desordenados. Cuando en una vuelta no se producen cambios el proceso termina. El seudocódigo del algoritmos es el siguiente:

```
BSCen:
vuelta=0
Repeat
   i = 1
   cambios=false
   while (i ≤ n−vuelta) do
      if a[i] > a[i + 1] then
                        swap(a[i], a[i + 1])
                        cambios=true
```

$i = i + 1$
vuelta=vuelta+1
Until (vuelta=$(n-1)$) or (not cambios)

El segundo algoritmo es *bubble sort con centinela por bloques* (BSBlo). El arreglo a ordenar se divide en bloques; cada bloque es ordenado utilizando bubble sort con centinela, y finalmente se realiza un merge de los bloques pre-ordenados. Seudocódigo:

BSBlo:
tamaño = $n$ div bloques
for $i = 1$ to bloques
BSCen
Merge

En el *Bubble sort paralelo* (BSPar) el arreglo se divide en bloques, y cada bloque es ordenado en paralelo por un proceso diferente usando bubble sort con centinela. Al terminar, cada proceso envía su porción a una tarea master que realiza el merge. Se trata de una variante del *Parallel Merge Sort*. El seudocódigo es el siguiente:

BSPar:
tamaño= $n$ div tareas
par $i = 1$ to tareas
Bscen
Merge

### 7.2.2 Pruebas iniciales

El trabajo o carga computacional que se realiza en estos algoritmos de sorting se encuentra básicamente influido, además del tamaño de la entrada o cantidad de elementos a ordenar, por el *grado de desorden* en que se encuentran los mismos.

En este sentido, el desorden se ve influido por el número de elementos que se encuentran fuera de su lugar final en la secuencia desordenada, el máximo desplazamiento de un elemento respecto de la que debe ser su posición final, y la manera en que se encuentran distribuidos los datos que están fuera de su lugar.

Luego, si $CargaDeTrabajo$ es la carga total de trabajo, $dmax$ es el máximo valor de las diferencias entre la posición actual de cada elemento y la posición final del mismo

en la secuencia ordenada, $CantDesordenados$ corresponde a la cantidad de elementos desordenados, y $DistrDesordenados$ se refiere a la forma en que están desordenados,

$$CargaDeTrabajo = f(dmax, CantDesordenados, DistrDesordenados)$$

En las pruebas con secuencias completas es sencillo encontrar la cantidad de elementos desordenados, ya que se obtiene contando (en una pasada) los datos ubicados en un lugar distinto al que les corresponde. Esto es sólo una parte del problema ya que la carga de trabajo también está influida por el número de movimientos que deben realizarse para llevar cada dato a su posición final.

Si se toma en cuenta sólo la suma de distancias de cada dato a su posición en la secuencia ordenada, el inconveniente es que se pierde información respecto de la cantidad de posiciones que hay que mover cada dato (el resultado de la sumatoria para mover $k$ números a distancia 1 es igual al de mover 1 número a distancia $k$, pero la carga de trabajo es diferente).

Con el objetivo de estudiar la influencia de los diferentes factores sobre la carga de trabajo se realizó experimentación sobre distintos tipos de secuencias de datos de entrada. En primer lugar se realizaron pruebas con secuencias de datos en las cuales las claves aparecen una única vez y el valor máximo de clave coincide con la cantidad de datos (*secuencias de datos completas*).

Las *pruebas de tipo 1* consisten en realizar la ordenación de un arreglo con sólo dos elementos intercambiados entre sí, y el resto en su posición final. En este caso el *desorden* está dado por el máximo desplazamiento. Por ejemplo, en la secuencia 8 2 3 4 5 6 7 1 9 10 el desorden es del 80%. Esto da una idea de la cantidad de iteraciones (y comparaciones) a realizar.

Las *pruebas de tipo 2* tratan con arreglos con dos bloques desordenados e invertidos. El *desorden* está dado por la cantidad de elementos desordenados, relacionado con la cantidad de *intercambios* a realizar. La cantidad de iteraciones se mantiene fija. Por ejemplo, en el arreglo 10 9 3 4 5 6 7 8 2 1 el desorden es del 40%.

Se utilizaron conjuntos de 100000, 300000, 500000 y 1000000 de datos, considerando desórdenes de 0 a 100% a intervalos de 10%.

### 7.2.3 Tiempos estimados y reales

#### BSCen

En el caso de secuencias de datos completos, el tiempo esperado para el bubble sort con centinela está dado por

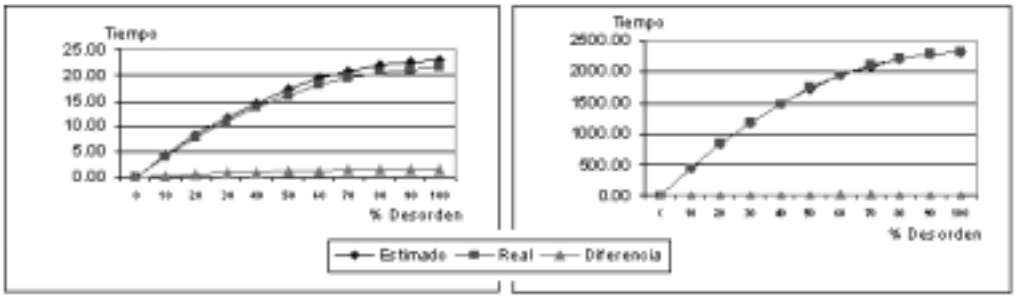


Figura 7.1: Gráfica de Tiempos Estimados y Reales para BSCen con secuencias de 100K y 1MB

$$TpoEstBSCen(n) = CantInt \times TpoInt + CantCo \times TpoCo \tag{7.1}$$

donde:

$CantInt = \sum_{i=1}^{n} \sum_{j=1}^{i} (a[i] < a[j])$ (cuenta para cada dato del arreglo cuántos valores ubicados en posiciones anteriores son mayores que él)

$CantCo = (dmax \times n) - ((dmax^2 + dmax)/2)$

$dmax = max$(posición actual de cada número $-$ posición final de ese número)

$TpoInt =$ tiempo para realizar un intercambio

$TpoCo =$ tiempo para realizar una comparación

Por otra parte, la carga de trabajo en el caso de BSCen tiene la forma:

$$TrabajoBSCen(n) = CantInt + CantCo \times Coef \tag{7.2}$$

donde $coef =$ relación entre entre el tiempo para realizar un intercambio y el tiempo para realizar una comparación.

Utilizando la Ec. 7.1 se calcularon los tiempos estimados para los casos mencionados en las pruebas y se midieron los obtenidos experimentalmente. La Tabla 7.1 muestra los valores estimados, reales y el valor absoluto de la diferencia entre ambos para secuencias de 100000 y 1000000 de datos en pruebas de tipo 1; puede observarse que la estimación es muy cercana a la realidad, especialmente para las secuencias de mayor tamaño. La Figura 7.1 presenta la gráfica correspondiente. Resultados similares se obtuvieron para tamaños de 300000 y 500000 datos, y en las pruebas de tipo 2.

| 100K | Desorden | T.Est | T.Real | Diferencia |
|---|---|---|---|---|
| | 0% | 4.60E-04 | 5.49E-04 | 8.93E-05 |
| | 10% | 4.37E+00 | 4.11E+00 | 2.56E-01 |
| | 20% | 8.27E+00 | 7.72E+00 | 5.55E-01 |
| | 30% | 1.17E+01 | 1.09E+01 | 7.86E-01 |
| | 40% | 1.47E+01 | 1.38E+01 | 9.15E-01 |
| | 50% | 1.72E+01 | 1.61E+01 | 1.13E+00 |
| | 60% | 1.93E+01 | 1.81E+01 | 1.16E+00 |
| | 70% | 2.09E+01 | 1.94E+01 | 1.46E+00 |
| | 80% | 2.21E+01 | 2.06E+01 | 1.48E+00 |
| | 90% | 2.27E+01 | 2.13E+01 | 1.47E+00 |
| | 100% | 2.30E+01 | 2.16E+01 | 1.39E+00 |
| **1MB** | **Desorden** | **T.Est** | **T.Real** | **Diferencia** |
| | 0% | 4.60E-03 | 4.72E-03 | 1.24E-04 |
| | 10% | 4.37E+02 | 4.40E+02 | 3.35E+00 |
| | 20% | 8.27E+02 | 8.31E+02 | 3.56E+00 |
| | 30% | 1.17E+03 | 1.17E+03 | 1.28E+00 |
| | 40% | 1.47E+03 | 1.48E+03 | 7.31E+00 |
| | 50% | 1.72E+03 | 1.73E+03 | 6.10E+00 |
| | 60% | 1.93E+03 | 1.95E+03 | 1.67E+01 |
| | 70% | 2.09E+03 | 2.11E+03 | 1.55E+01 |
| | 80% | 2.21E+03 | 2.20E+03 | 2.26E+00 |
| | 90% | 2.27E+03 | 2.28E+03 | 9.63E+00 |
| | 100% | 2.30E+03 | 2.31E+03 | 1.20E+01 |

Tabla 7.1: Tiempos Estimados y Reales para BSCen con secuencias de 100K y 1MB

### BSBlo

El tiempo estimado para el bubble sort con centinela por bloques, dado que se trata de un algoritmo secuencial, es la suma de los tiempos para ordenar cada uno de los bloques y el tiempo finales para realizar la mezcla de las subsecuencias ordenadas. Si $k$ es el número de bloques en que se divide la secuencia, entonces

$$TpoEstBSBlo(n,k) = \sum_{q=1}^{k} TpoEstBSCen(n/k) + TpoMerge \tag{7.3}$$

donde $TpoMerge$ = es el tiempo para realizar la mezcla de los bloques.

También, en este caso utilizando la Ec. 7.3, se calcularon los tiempos estimados y se los comparó con los reales obtenidos experimentalmente. Se verificó una correspondencia similar a lo observado para BSCen; se utilizaron 4, 8, 16, 32, 64 y 128 bloques.

La mejora en tiempo del algoritmo BSBlo respecto de BSCen puede verse en la Figura 7.2 para secuencias de 1MB en pruebas de tipo 1 (resultados análogos se obtienen para secuencias de 100, 300 y 500 mil datos, y pruebas de tipo 2). La Tabla 7.2 muestra los tiempos de BSCen, BSBlo y la relación BSBlo/BSCen con 4, 8 y 16 bloques (Rel(k)) para secuencias de 1000000 de datos en pruebas de tipo 1. La Tabla 7.3 comprende los mismos datos pero para secuencias totalmente ordenadas inicialmente (Ale0) y distintas

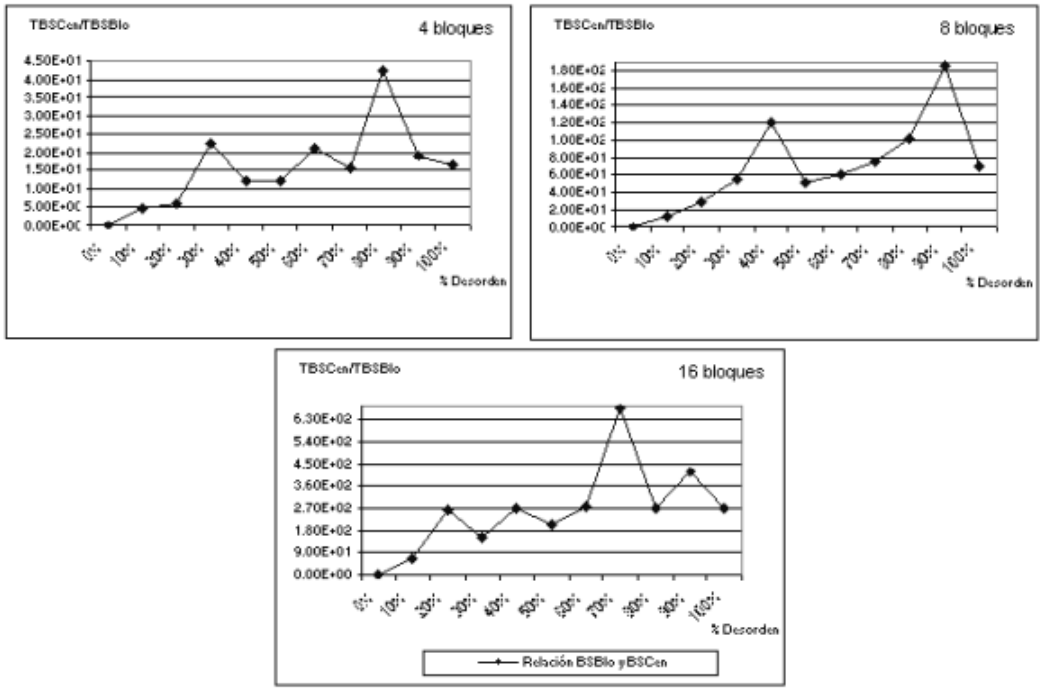


Figura 7.2: Relación entre los tiempos de BSBlo y BSCen en secuencias de 1MB (pruebas de tipo 1) con 4, 8 y 16 bloques

"seudo-aleatorias" (secuencias aleatorias levemente modificadas).

### BSPar

En este caso el tiempo estimado está dado por el tiempo de la tarea que más tarda para ordenar su bloque sumado a la comunicación y el merge final. Si $k$ es el número de bloques en que se divide la secuencia, entonces el tiempo estimado para BSPar es

$$TpoEstBSPar(n,k) = (CantIntP \times TpoInt + CantCoP \times TpoCo) + TpoComunic + TpoMerge$$

donde:

$CantIntP = \sum_{i=1}^{n/k} \sum_{j=1}^{i} (a[i] < a[j])$ (cuenta para cada dato del bloque cuántos valores ubicados en posiciones anteriores son mayores que él)

$CantCoP = (dmax \times (n/k)) - ((dmax^2 + dmax)/2)$

$dmax = max$(posición actual de cada número $-$ posición final de ese número)

$TpoInt =$ tiempo para realizar un intercambio

| Desorden | BSCen | BSBlo(4) | Rel(4) | BSBlo(8) | Rel(8) | BSBlo(16) | Rel(16) |
|---|---|---|---|---|---|---|---|
| 0% | 4.72E-03 | 5.22E-02 | 9.04E-02 | 9.22E-02 | 5.12E-02 | 1.49E-01 | 3.17E-02 |
| 10% | 4.40E+02 | 9.21E+01 | 4.78E+00 | 3.30E+01 | 1.33E+01 | 7.14E+00 | 6.16E+01 |
| 20% | 8.31E+02 | 1.36E+02 | 6.10E+00 | 2.84E+01 | 2.92E+01 | 3.12E+00 | 2.66E+02 |
| 30% | 1.17E+03 | 5.23E+01 | 2.24E+01 | 2.17E+01 | 5.40E+01 | 8.11E+00 | 1.44E+02 |
| 40% | 1.48E+03 | 1.20E+02 | 1.23E+01 | 1.23E+01 | 1.20E+02 | 5.44E+00 | 2.72E+02 |
| 50% | 1.73E+03 | 1.42E+02 | 1.22E+01 | 3.34E+01 | 5.17E+01 | 8.46E+00 | 2.04E+02 |
| 60% | 1.95E+03 | 9.26E+01 | 2.10E+01 | 3.21E+01 | 6.07E+01 | 7.11E+00 | 2.74E+02 |
| 70% | 2.11E+03 | 1.36E+02 | 1.55E+01 | 2.83E+01 | 7.43E+01 | 3.13E+00 | 6.74E+02 |
| 80% | 2.20E+03 | 5.21E+01 | 4.23E+01 | 2.15E+01 | 1.02E+02 | 8.13E+00 | 2.71E+02 |
| 90% | 2.28E+03 | 1.20E+02 | 1.91E+01 | 1.23E+01 | 1.86E+02 | 5.47E+00 | 4.18E+02 |
| 100% | 2.31E+03 | 1.41E+02 | 1.64E+01 | 3.35E+01 | 6.90E+01 | 8.58E+00 | 2.69E+02 |

Tabla 7.2: Tiempos BSCen, BSBlo y BSBlo/BSCen para 4, 8 y 16 bloques en secuencias de 1 MB (pruebas de tipo 1)

| Secuencia | BSCen | BSBlo(4) | Rel(4) | BSBlo(8) | Rel(8) | BSBlo(16) | Rel(16) |
|---|---|---|---|---|---|---|---|
| Ale0 | 4.79E-03 | 5.31E-02 | 9.02E-02 | 7.97E-02 | 6.01E-02 | 1.46E-01 | 3.27E-02 |
| Ale1 | 2.45E+03 | 1.56E+03 | 1.57E+00 | 8.38E+02 | 2.92E+00 | 4.35E+02 | 5.62E+00 |
| Ale2 | 5.24E+03 | 1.74E+03 | 3.01E+00 | 8.78E+02 | 5.97E+00 | 4.43E+02 | 1.18E+01 |
| Ale3 | 6.72E+03 | 1.74E+03 | 3.86E+00 | 8.54E+02 | 7.87E+00 | 4.26E+02 | 1.58E+01 |
| Ale4 | 6.71E+03 | 1.66E+03 | 4.05E+00 | 8.14E+02 | 8.24E+00 | 4.03E+02 | 1.67E+01 |

Tabla 7.3: Tiempos BSCen, BSBlo y BSBlo/BSCen para 4, 8 y 16 bloques en secuencias "seudo-aleatorias" de 1 MB

$TpoCo$ = tiempo para realizar una comparación

$TpoComunic$ = tiempo utilizado en comunicaciones

$TpoMerge$ = tiempo para realizar la mezcla de los bloques

Para calcular el tiempo de intercambio y comparaciones se utilizaron los datos de la tarea que más tiempo tardó.

El trabajo se calcula como

$$TrabajoBSPar(n,k) = CantIntP + CantCoP \times Coef$$

También en este caso se toma en cuenta a la "peor" tarea. No se incluye en el cálculo del trabajo al tiempo de comunicación (depende del mecanismo utilizado) ni el tiempo para realizar el merge (de orden fijo).

En estas pruebas se utilizaron 4, 8 y 16 procesos. La Figura 7.3 muestra el "speedup" (dado por el cociente del tiempo de BSBlo y BSPar) y la "eficiencia" (relación entre dicho cociente y el número de tareas) para secuencias de 1MB en pruebas de tipo 2 (resultados análogos se obtienen para secuencias de 100, 300 y 500 mil datos). En el caso de las pruebas de tipo 1, la ganancia en tiempo no es tan notoria, debido a que el trabajo a realizar es menor.

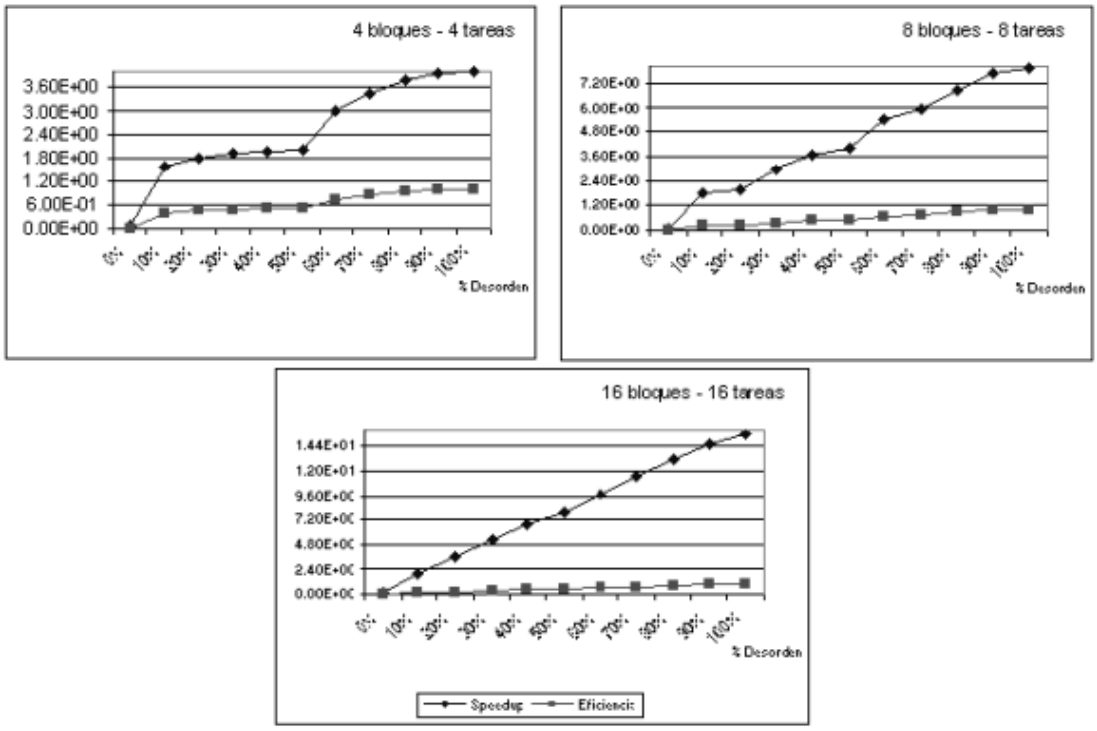


Figura 7.3: Relación entre BSBlo y BSPar en secuencias de 1MB (pruebas de tipo 2) con 4, 8 y 16 bloques/tareas (Speedup y Eficiencia)

La Tabla 7.4 muestra los tiempos de BSBlo, BSPar, la relación BSBlo/BSPar con $k$ =8 y 16 bloques (Speedup(k)) y las eficiencias correspondientes, para secuencias de 1000000 de datos en pruebas de tipo 2. La Tabla 7.5 comprende los mismos datos pero para secuencias totalmente ordenadas inicialmente (Ale0) y distintas "seudo-aleatorias" (secuencias aleatorias levemente modificadas).

### Algoritmo de Merge utilizado

Se comparó la implementación del Merge mediante dos métodos diferentes:

- Multifases, consistente en realizar mezclas parciales de a pares de bloques en fases sucesivas, obteniendo en cada etapa subsecuencias del doble de tamaño.
- Multibloques, en el cual la mezcla se realiza en una única etapa con todos los bloques a la vez.

El tiempo de ejecución de la mezcla depende sólo del tamaño de la entrada, no del grado de desorden que presentan los datos. El resultado en tiempo de ejecución fue más

| Desorden | BSBlo(8) | BSPar(8) | Sp(8) | E(8) | BSBlo(16) | BSPar(16) | Sp(16) | E(16) |
|---|---|---|---|---|---|---|---|---|
| 0% | 7.97E-02 | 7.06E-01 | 1.13E-01 | 1.41E-02 | 1.46E-01 | 7.82E-01 | 1.87E-01 | 1.17E-02 |
| 10% | 1.43E+02 | 8.00E+01 | 1.78E+00 | 2.23E-01 | 4.89E+01 | 2.50E+01 | 1.95E+00 | 1.22E-01 |
| 20% | 1.97E+02 | 1.01E+02 | 1.96E+00 | 2.45E-01 | 9.41E+01 | 2.56E+01 | 3.67E+00 | 2.30E-01 |
| 30% | 2.98E+02 | 1.02E+02 | 2.91E+00 | 3.64E-01 | 1.36E+02 | 2.56E+01 | 5.32E+00 | 3.33E-01 |
| 40% | 3.80E+02 | 1.03E+02 | 3.70E+00 | 4.62E-01 | 1.75E+02 | 2.57E+01 | 6.81E+00 | 4.26E-01 |
| 50% | 4.07E+02 | 1.03E+02 | 3.96E+00 | 4.95E-01 | 2.02E+02 | 2.57E+01 | 7.84E+00 | 4.90E-01 |
| 60% | 5.49E+02 | 1.03E+02 | 5.35E+00 | 6.69E-01 | 2.50E+02 | 2.57E+01 | 9.74E+00 | 6.09E-01 |
| 70% | 6.04E+02 | 1.03E+02 | 5.88E+00 | 7.35E-01 | 2.96E+02 | 2.58E+01 | 1.15E+01 | 7.17E-01 |
| 80% | 7.05E+02 | 1.03E+02 | 6.86E+00 | 8.58E-01 | 3.38E+02 | 2.58E+01 | 1.31E+01 | 8.18E-01 |
| 90% | 7.87E+02 | 1.03E+02 | 7.66E+00 | 9.58E-01 | 3.76E+02 | 2.58E+01 | 1.46E+01 | 9.10E-01 |
| 100% | 8.14E+02 | 1.03E+02 | 7.92E+00 | 9.90E-01 | 4.03E+02 | 2.59E+01 | 1.56E+01 | 9.73E-01 |

Tabla 7.4: Tiempos BSBlo, BSPar, Speedup y Eficiencia para 8 y 16 bloques/tareas en secuencias de 1 MB (pruebas de tipo 2)

| Secuencia | BSBlo(8) | BSPar(8) | Sp(8) | E(8) | BSBlo(16) | BSPar(16) | Sp(16) | E(16) |
|---|---|---|---|---|---|---|---|---|
| Ale0 | 7.97E-02 | 7.06E-01 | 1.13E-01 | 1.41E-02 | 1.46E-01 | 7.82E-01 | 1.87E-01 | 1.17E-02 |
| Ale1 | 8.38E+02 | 1.09E+02 | 7.66E+00 | 9.58E-01 | 4.35E+02 | 2.90E+01 | 1.50E+01 | 9.39E-01 |
| Ale2 | 8.78E+02 | 1.16E+02 | 7.58E+00 | 9.47E-01 | 4.43E+02 | 2.92E+01 | 1.52E+01 | 9.49E-01 |
| Ale3 | 8.54E+02 | 1.15E+02 | 7.46E+00 | 9.32E-01 | 4.26E+02 | 2.89E+01 | 1.47E+01 | 9.21E-01 |
| Ale4 | 8.14E+02 | 1.03E+02 | 7.92E+00 | 9.90E-01 | 4.03E+02 | 2.59E+01 | 1.56E+01 | 9.73E-01 |

Tabla 7.5: Tiempos BSBlo, BSPar, Speedup y Eficiencia para 8 y 16 bloques/tareas en secuencias "seudo-aleatorias" de 1 MB

satisfactorio en el caso del Merge Multibloques, básicamente por el modelo de arquitectura utilizado, por lo que fue el utilizado para la experimentación.

La Figura 7.4 muestra los resultados de la comparación de tiempos entre ambos métodos para secuencias de 10K, 100K, 1MB y 10MB, con 4, 8 y 16 bloques/tareas. La Tabla 7.6 muestra los tiempos de los Merge Multibloques y Multifases, para los mismos casos.

### 7.2.4 Relación entre la carga de trabajo y el grado de desorden

El tiempo de ejecución de los algoritmos de sorting presentados se encuentra directamente relacionado con la *carga de trabajo* en cada caso. Por otra parte, esta carga se ve influida por el *grado de desorden* de los datos.

La carga de trabajo a realizar en términos de comparaciones e intercambios puede conocerse estáticamente a través de un cálculo basado en las posiciones iniciales de las claves,. En el caso de secuencias completas el tiempo necesario para conocer este valor es $O(n^2)$; en el caso general es $n^2$.

Puede definirse el grado de desorden como un porcentaje del trabajo a realizar respecto del mejor y peor caso (secuencias totalmente ordenadas o totalmente invertidas). Para

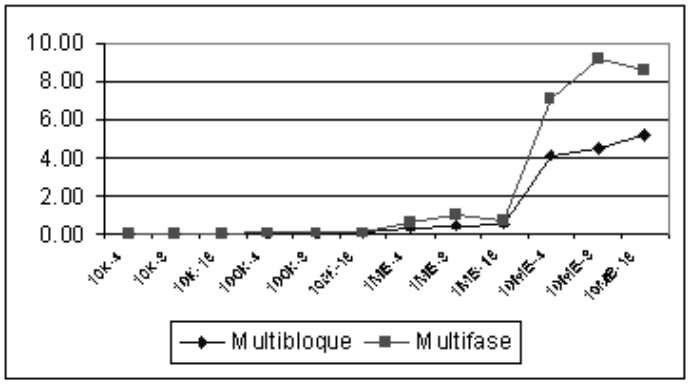


Figura 7.4: Comparación entre Merge Multifases y Multibloques

| Tamaño | Tareas | Multibloques | Multifases |
|---|---|---|---|
| 10K | 4 | 4.65E-03 | 8.00E-03 |
| 10K | 8 | 6.85E-03 | 8.97E-03 |
| 10K6 | 16 | 7.03E-03 | 9.75E-03 |
| 100K | 4 | 4.32E-02 | 7.14E-02 |
| 100K | 8 | 4.43E-02 | 7.73E-02 |
| 100K | 16 | 5.71E-02 | 8.17E-02 |
| 1MB | 4 | 4.21E-01 | 7.04E-01 |
| 1MB | 8 | 4.58E-01 | 1.04E+00 |
| 1MB | 16 | 5.42E-01 | 7.95E-01 |
| 10MB | 4 | 4.13E+00 | 7.07E+00 |
| 10MB | 8 | 4.50E+00 | 9.19E+00 |
| 10MB | 16 | 5.15E+00 | 8.57E+00 |

Tabla 7.6: Tiempos de Merge Multifases y Multibloques

| Tipo 1 | % Desorden | Gr.Desorden | Tpo.BSCen | Tipo 2 | % Desorden | Gr.Desorden | Tpo.BSCen |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 4.72E-03 | | 0 | 0 | 4.79E-03 |
| | 10 | 13 | 4.40E+02 | | 10 | 73 | 3.27E+03 |
| | 20 | 24 | 8.31E+02 | | 20 | 78 | 4.12E+03 |
| | 30 | 34 | 1.17E+03 | | 30 | 83 | 4.83E+03 |
| | 40 | 42 | 1.48E+03 | | 40 | 88 | 5.36E+03 |
| | 50 | 50 | 1.73E+03 | | 50 | 92 | 5.77E+03 |
| | 60 | 56 | 1.95E+03 | | 60 | 95 | 6.14E+03 |
| | 70 | 60 | 2.11E+03 | | 70 | 97 | 6.41E+03 |
| | 80 | 64 | 2.20E+03 | | 80 | 99 | 6.60E+03 |
| | 90 | 66 | 2.28E+03 | | 90 | 100 | 6.69E+03 |
| | 100 | 66 | 2.31E+03 | | 100 | 100 | 6.71E+03 |

Tabla 7.7: Tiempos BSCen para distintos Grados de Desorden en secuencias de 1 MB

una secuencia dada *sec*,

$$GradoDesorden(sec) = \frac{(Trabajo(sec) - TrabajoMinimo) \times 100}{TrabajoMaximo - TrabajoMinimo}$$

donde

$Trabajo(sec)$ = trabajo necesario para ordenar la secuencia *sec* usando BSCen,

$TrabajoMinimo$ = trabajo realizado en el mejor caso para secuencias del tamaño de *sec*

$TrabajoMaximo$ = trabajo realizado en el peor caso para secuencias del tamaño de *sec*

Es claro que un menor grado de desorden resulta en una reducción del tiempo de ordenación. Se realizó abundante experimentación consistente en obtener, para secuencias con diferentes grados de desorden, el tiempo de ejecución del sorting usando BSCen. La Tabla 7.7 presenta los tiempos para distintos grados de desorden en secuencias de 1MB; se muestra también la relación entre el porcentaje de desorden utilizado en las pruebas de tipo 1 y 2 y el grado de desorden definido. En la Figura 7.5 puede observarse la gráfica correspondiente.

Luego, se dividieron las secuencias en distintos bloques para obtener el tiempo de ordenación de cada bloque y el grado de desorden en cada uno de ellos. Sea $S$ el cociente entre el tiempo secuencial usando BSCen y el tiempo paralelo usando BSPar, y $D$ el cociente entre el porcentaje de desorden de la secuencia completa y el mayor porcentaje de desorden en una tarea. La Figura 7.6 muestra que $S$ es proporcional a la relación entre el porcentaje de desorden total y el de la peor tarea, para secuencias de tipo 1, de tipo 2 y seudo-aleatorias de 1MB (para mayor claridad del gráfico, los valores de $D$ en cada caso fueron multiplicados por una constante).

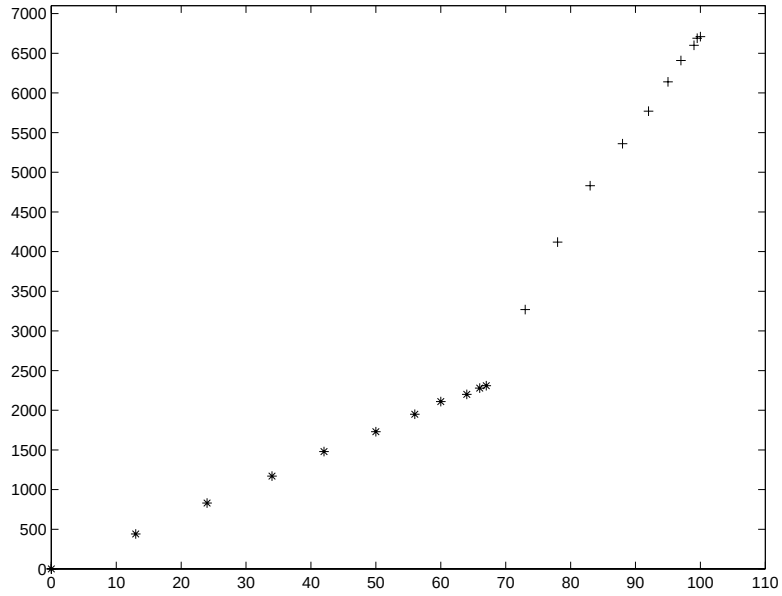


Figura 7.5: Tiempo BSCen para diferentes *grados de desorden*

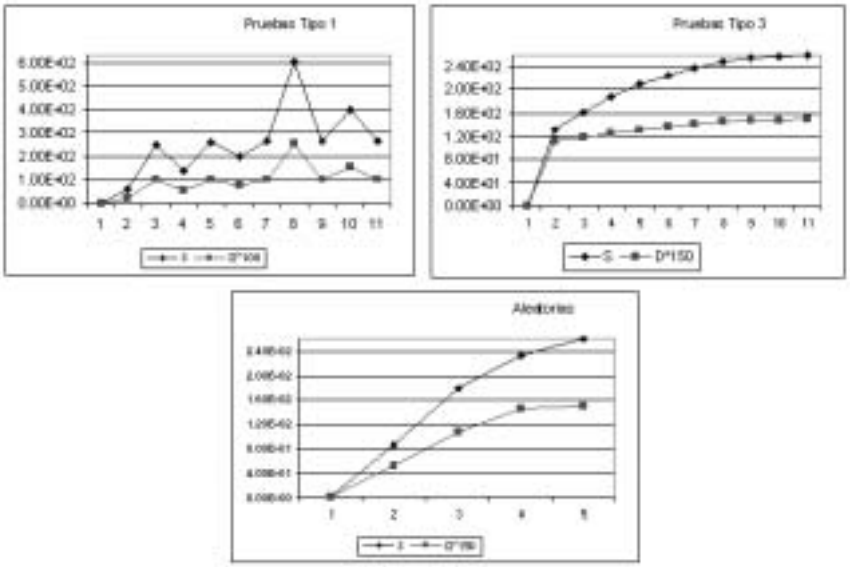


Figura 7.6: Relación entre $S$ y $D$ para secuencias de 1MB

## 7.3 Balance de la Carga de Trabajo

Como se expresó, el problema de sorting tiene la característica de que la carga de trabajo no depende sólo del tamaño de la entrada sino que influyen otros factores (grado de desorden). Si bien es posible conocer *a priori* el trabajo, la complejidad de $O(n^2)$ para obtener dicho valor lo convierte en poco práctico para secuencias de datos de gran tamaño.

Cuando el problema es resuelto en forma paralela, dividiendo el espacio de datos entre varias tareas, la carga de trabajo (y por consiguiente el tiempo) en cada una también depende del grado de desorden de los datos. Por esta razón, para obtener buenos tiempos de ejecución es necesario que la carga en cada tarea sea cercana al punto de equilibrio de trabajo por tarea.

Para conseguir el objetivo de lograr carga de trabajo balanceada, el método propuesto utiliza información propia del problema para distribuir los datos equitativamente no respecto del tamaño sino del trabajo que producen.

### 7.3.1 Evolución de la Carga de trabajo

Con el objetivo de analizar en detalle el trabajo realizado por BSCen se estudió el mismo en diferentes momentos de la ejecución. No se utilizaron las secuencias *completas* sino archivos con *datos al azar con una distribución normal* (considerando éste como el caso más standard o probable).

En particular, se midió el trabajo a intervalos regulares de iteraciones: si $NroIter$ es el total de iteraciones necesarias para ordenar una secuencia, entonces se observó el trabajo realizado a diferentes porcentajes de $NroIter$. La carga total de trabajo es la suma de los trabajos parciales en cada intervalo:

$$CargaTotalTrabajo = \sum_{r=1}^{h} Trabajo_r \tag{7.4}$$

donde $Trabajo_r$ = es el trabajo realizado en el intervalo $r$ (función de la cantidad de comparaciones e intercambios) y,

$h$ es la cantidad de intervalos considerados, que depende de la "frecuencia de muestreo" elegida (si se mide el trabajo cada $U\%$ de las iteraciones, entonces $h = 100/U$).

La Figura 7.7 muestra los resultados de trabajo promedio usando BSCen para 100 archivos con la característica descripta, tomando el trabajo cada 10% de las iteraciones. Puede observarse que el trabajo realizado en cada intervalo considerado decrece a medida que el algoritmo avanza en su ejecución. La Tabla 7.8 muestra el trabajo acumulado y el

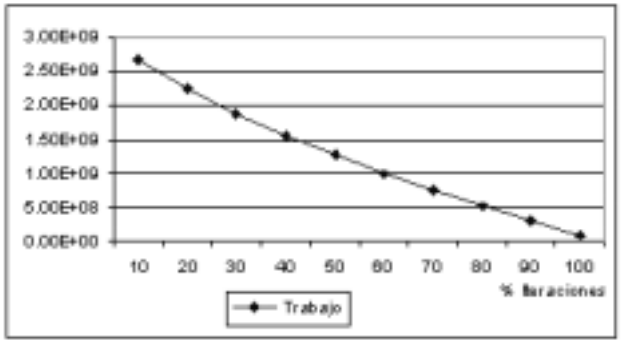


Figura 7.7: Trabajo realizado cada 10% de las iteraciones para secuencias al azar de 1MB

| % Iteraciones | Trabajo Acumulado | Trabajo en el intervalo |
|---|---|---|
| 10 | 2.672E+09 | 2.672E+09 |
| 20 | 4.906E+09 | 2.234E+09 |
| 30 | 6.781E+09 | 1.875E+09 |
| 40 | 8.338E+09 | 1.557E+09 |
| 50 | 9.607E+09 | 1.269E+09 |
| 60 | 1.061E+10 | 1.004E+09 |
| 70 | 1.137E+10 | 7.564E+08 |
| 80 | 1.189E+10 | 5.246E+08 |
| 90 | 1.220E+10 | 3.061E+08 |
| 100 | 1.230E+10 | 9.891E+07 |

Tabla 7.8: Trabajo cada 10% de las iteraciones

realizado cada 10% de las iteraciones.

Por otra parte, puede estudiarse qué porcentaje del trabajo total se realiza en cada intervalo de las iteraciones consideradas. En este sentido, la Figura 7.8 muestra la función acumulativa del porcentaje de trabajo realizado en cada intervalo para secuencias al azar de 1MB, mientras la Tabla 7.9 muestra los valores correspondientes.

| % Iteraciones | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|
| % Trabajo intervalo | 11.3784 | 21.7266 | 31.2041 | 39.8951 | 47.8593 | 55.1416 | 61.7796 | 67.8055 | 74.9436 | 78.1276 |
| % Iteraciones | 55 | 60 | 65 | 70 | 75 | 80 | 85 | 90 | 95 | 100 |
| % Trabajo intervalo | 83.7953 | 86.2895 | 89.9746 | 92.4406 | 94.8019 | 96.7063 | 98.2537 | 99.1957 | 99.7836 | 100.0000 |

Tabla 7.9: Porcentajes de trabajo realizado cada 5% de las iteraciones, en secuencias al azar de 1MB

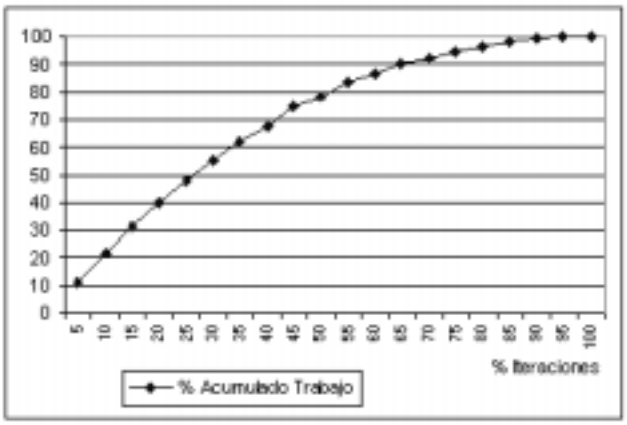


Figura 7.8: Acumulativa del porcentaje de trabajo cada 5% de las iteraciones para secuencias al azar de 1MB

### 7.3.2 Estimación

El análisis matemático del desempeño de los algoritmos de sorting se da en el contexto del estudio de las propiedades básicas de las permutaciones. El detalle de tales estudios excede el marco de esta Tesis [1].

El valor medio del trabajo puede tomarse como un *estimador* razonable de la carga total de trabajo a realizar por el algoritmo, obteniéndose buenos resultados. La idea es la siguiente: si se cuenta con estimadores del trabajo total y del trabajo en cada intervalo, pueden utilizarse para *predecir*, al terminar un intervalo, cuál es la carga de trabajo restante. Reescribiendo la Ec. 7.4,

$$CargaTotalTrabajo = \sum_{r=1}^{g} Trabajo_r + \sum_{r=g+1}^{h} Trabajo_r$$

Luego, al finalizar $g$ intervalos, el trabajo restante puede escribirse como

[1] Referencias clásicas en este tema son [203, 301]. El trabajo realizado por el algoritmo bubble sort *sin centinela*, cuando los datos son al azar, sigue una distribución muy cercana a una Normal más una constante. El valor de la constante es $(n \times (n-1) \times 2)$, la media de la Normal es $n \times (n-1)/4$ y el valor medio del tiempo de ejecución es $(3/4) \times n \times (n-1)$. El índice de dispersión de la Normal (desvío standard / valor medio) decrece cuando el tamaño de la secuencia se incrementa. En el caso del bubble sort con centinela, puede verse que hace $\sim n^2/2$ comparaciones y mueve $\sim n^2/2$ registros en promedio en $\sim n - \sqrt{\pi n/2}$ pasadas, para datos al azar. En la bibliografía no se encuentran resultados analíticos de otras características de las distribuciones de las variables aleatorias involucradas, tales como varianzas o momentos de orden superior. Por este motivo en este trabajo se ha recurrido a la estimación de esas características usando simulación de MonteCarlo

| % Iteraciones | Trab.Estimado | Trab.Real | Diferencia | % Diferencia |
|---|---|---|---|---|
| 10% | 12297184332 | 12297186280 | 1947.529808 | 1.58372E-05 |
| 15% | 12297184575 | 12297186280 | 1704.731134 | 1.38628E-05 |
| 20% | 12297184808 | 12297186280 | 1471.968744 | 1.197E-05 |
| 25% | 12297184996 | 12297186280 | 1284.391672 | 1.04446E-05 |
| 30% | 12297185186 | 12297186280 | 1093.910896 | 8.89562E-06 |
| 35% | 12297185382 | 12297186280 | 897.5907822 | 7.29916E-06 |
| 40% | 12297185597 | 12297186280 | 682.6073799 | 5.55092E-06 |
| 50% | 12297185906 | 12297186280 | 374.3361397 | 3.04408E-06 |
| 60% | 12297186117 | 12297186280 | 162.6018124 | 1.32227E-06 |
| 70% | 12297186221 | 12297186280 | 59.15789032 | 4.81069E-07 |
| 75% | 12297186250 | 12297186280 | 29.90008354 | 2.43146E-07 |
| 80% | 12297186269 | 12297186280 | 11.43418503 | 9.29821E-08 |
| 90% | 12297186275 | 12297186280 | 4.759670258 | 3.87054E-08 |
| 100% | 12297186280 | 12297186280 | 0 | 0 |

Tabla 7.10: Trabajo Estimado y Real en secuencias al azar de 1MB, en distintos porcentajes de iteraciones

$$CargaRestante = CargaTotalTrabajo - \sum_{r=1}^{g} Trabajo_r \tag{7.5}$$

Luego, si en lugar de los valores de trabajo reales se utilizan los valores estimados, la Ec. 7.5 se convierte en

$$CargaRestanteEstimada \approx CargaTotalTrabajoEstimado - \sum_{r=1}^{g} Trabajo_r$$

Intuitivamente, a medida que $g \to h$ la estimación es más precisa. En el caso extremo, cuando $g = h$ se conoce el trabajo con exactitud. La pregunta entonces es: *en qué momento se puede tener un estimador aceptable del trabajo restante*?

Se estudió, a distintos porcentajes de iteraciones, cuál era la diferencia entre el trabajo total estimado a partir de lo realizado en ese porcentaje y el trabajo real que debía realizarse para ordenar la secuencia de datos. La Figura 7.9(a) muestra el trabajo real y el estimado en distintos porcentajes de vueltas (en promedio, para 100 secuencias al azar de 1MB). La Figura 7.9(b) presenta el porcentaje de diferencia entre el trabajo estimado y el real (en promedio, para las mismas secuencias); este valor se calcula como $ABS(TrabajoEstimado - TrabajoReal) \times 100/TrabajoReal$. En la Figura 7.9(c) pueden observarse los porcentajes de diferencia entre trabajo real y estimado usando el 5% de las iteraciones para 50 secuencias al azar de 1MB. La Tabla 7.10 muestra los valores correspondientes.

En particular, puede observarse que considerar el 5% de las iteraciones puede ser suficiente para estimar el trabajo que realizaría cada proceso. Este 5% de las vueltas representa aproximadamente el 11% del trabajo de las vueltas. Esta consideración se

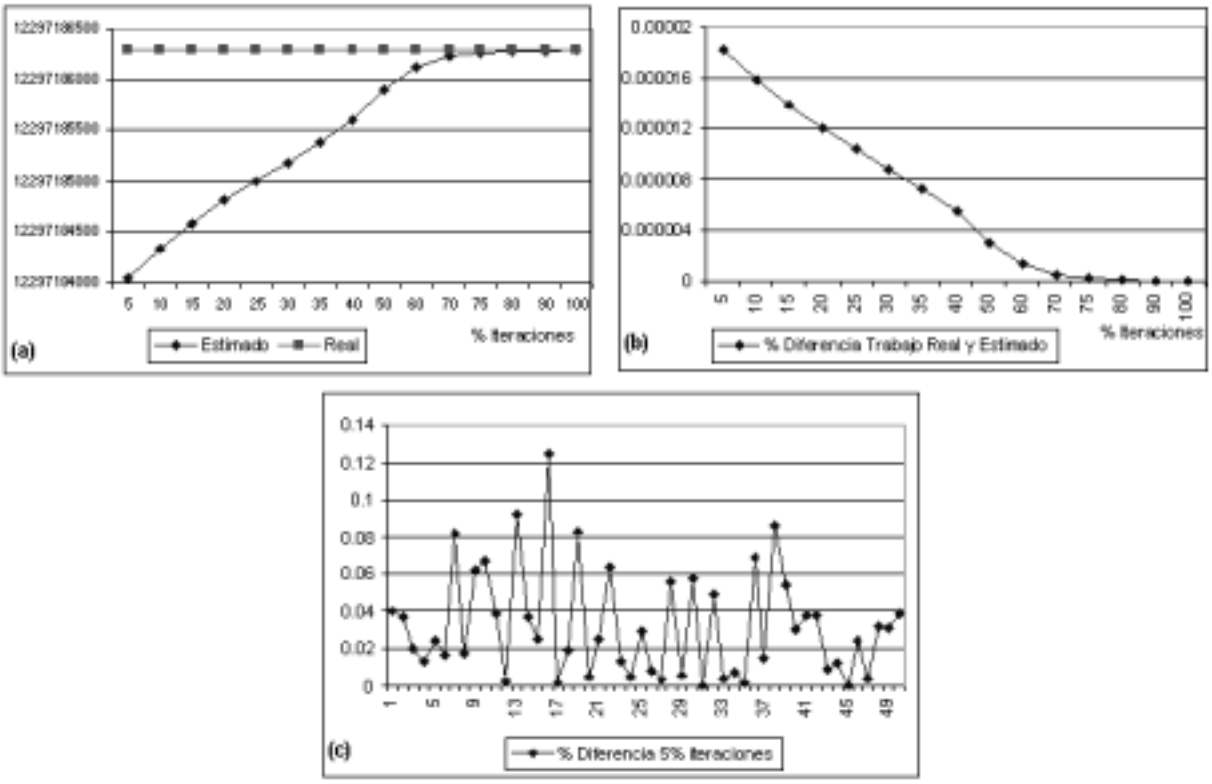


Figura 7.9: Trabajo Real y Estimado en secuencias al azar de 1MB. (a) En distintos porcentajes de iteraciones. (b) Porcentaje de diferencia. (c) Porcentaje de diferencia usando el 5% de iteraciones

basa en que el desvío standard obtenido es considerablemente pequeño (alrededor del $0.5 \times 10^{-3}$).

Con el objetivo de tener en cuenta situaciones en las cuales la secuencia de datos no fuera al azar con distribución normal, se experimentó con secuencias en las que los datos se encuentran totalmente invertidos. Siguiendo un razonamiento similar al del caso de secuencias al azar, el 5% de las iteraciones representan aproximadamente el 9% del trabajo total.

A partir de este resultado, si con el trabajo realizado hasta el porcentaje de iteraciones elegido se considera que no se encuentra dentro de los parámetros de una muestra aleatoria, se realiza una interpolación entre el valor estimado para el caso al azar y el peor caso. Luego, el trabajo realizado hasta ese momento va a representar un porcentaje del total que estará entre 9.5 y 11.

Entonces, puede estimarse el trabajo que resta realizar luego del 5% de las vueltas como

$$TrabajoRestanteEstimado \approx TrabajoRealizado \times 100/CoefTra$$

donde $CoefTra$ es un valor que varía entre 9.5 y 11, dependiendo de si el trabajo realizado se encuentra dentro de los parámetros normales o no.

## 7.4 Método de Sorting Paralelo con Balance de Carga Dinámico (*SPBCD*)

Esta sección presenta un método de sorting paralelo que balancea dinámicamente la carga utilizando un *estimador* que permite *predecir* el trabajo pendiente. La noción principal es la de distribuir inicialmente entre las tareas sólo un porcentaje del trabajo total a realizar; luego de un período de tiempo, conociendo lo realizado por cada tarea, se estima el trabajo restante en cada una y se distribuye la porción restante entre ellas con el objetivo de que realicen un trabajo equitativo.

En la sección anterior se observó que luego del 5% de las iteraciones se tiene un estimador razonable del trabajo realizado, y con él se puede predecir el trabajo restante. El esquema planteado aquí se basa en este hecho para realizar la ordenación de una secuencia de datos mediante un algoritmo que balancea dinámicamente la carga.

Resulta claro que cuanto mayor sea el porcentaje de iteraciones considerado, la estimación del trabajo restante será más precisa. Pero, en casos donde las secuencias a ordenar por cada tarea son muy dispares en cuanto a su grado de desorden, el hecho de

demorar el momento para realizar la estimación puede resultar en esperas por parte de alguno de los procesos.

### 7.4.1 Método de distribución dinámica propuesto

Dadas una secuencia de $n$ datos a ordenar, un proceso o tarea *maestra*, y $k$ procesos o tareas *esclavas*, el método presentado (*SPBCD*) consiste en:

1. Repartir un porcentaje de los datos (*bloque1*) en forma equitativa a cada uno de las $k$ tareas esclavas, dejando el resto de los datos sin distribuir.
2. Cuando los procesos han completado el 5% de las iteraciones de su bloque (con respecto a la cantidad media de iteraciones a realizar de acuerdo al tamaño del bloque asignado), estimar el trabajo total a realizar por cada proceso.
3. Con la estimación obtenida, distribuir el resto de los datos (*bloque2*) tratando de balancear la carga de trabajo total que realizará cada proceso. El trabajo total de un proceso es la suma del trabajo realizado para ordenar el bloque de tamaño *bloque1* /$k$ y el segundo bloque (fracción de *bloque2*).
4. Reunir las subsecuencias ordenadas por las tareas esclavas y realizar el merge de las mismas.

El método propuesto se encuadra en los esquemas de algoritmos de balance de carga dinámicos centralizados, utilizando el paradigma de computación paralela *master/slave*. Fue comparado básicamente (respecto del balance de carga y el tiempo de ejecución) contra otras dos variantes:

- Sorting Paralelo Sin Redistribución (*SPSR*): el total de los datos es distribuido en forma equitativa (con respecto al tamaño de bloque) a cada proceso.
- Sorting Paralelo Con Distribución Fija (*SPCDF*): se reparte un porcentaje de los datos en forma equitativa a cada uno de los procesos y luego se distribuye el resto de la misma forma.

En secciones posteriores se analizan los resultados de estas comparaciones.

### 7.4.2 Seudocódigo

En esta sección se describe el seudocódigo del método, además de dar algunas precisiones en cuanto a la implementación del mismo.

### Tarea Maestra

Maestra:
- Repartir un porcentaje de los datos
- Recibir trabajo realizado por cada tarea luego del 5% de las iteraciones
- Estimar el trabajo restante para cada proceso
- Distribuir los datos reservados inicialmente
- Recibir resultado ordenado de cada proceso
- Realizar el merge

### Tarea Esclava

Las ordenaciones de los bloques se realizan utilizando BSCen.

Esclava:
- Recibir los datos a ordenar en la primera parte
- Ordenar los datos hasta completar el 5% de las iteraciones
- Enviar la cantidad de trabajo realizado a la tarea Maestra
- Completar la ordenacion
- Recibir trabajo restante
- Ordenar nuevo bloque
- Realizar el merge de los dos bloques
- Enviar el resultado a la tarea Maestra

### Precisiones

*Repartir un porcentaje de los datos*

El porcentaje a repartir se calcula en función de un parámetro del algoritmo. En las pruebas realizadas se varió entre 0% (indicando el caso extremo en que se distribuyen inicialmente todos los valores) y 40% (reparto inicial del 60% de los datos, dejando un 40% para distribuir dinmicamente). A cada tarea se le envía el mismo tamaño de bloque.

*Recibir trabajo realizado por cada tarea luego del 5% de las iteraciones*

El 5% de las iteraciones es calculado respecto del peor caso para el tamaño de bloque correspondiente.

*Estimar el trabajo restante para cada proceso*

Sean $CantE$ = cantidad de datos que recibió originalmente una tarea esclava, $TrabajoP[i]$ = trabajo realizado por la tarea $i$ en el 5% de las iteraciones, y $TrabajoT[i]$ = trabajo total estimado para la tarea $i$ en el primer bloque.

Las estimaciones son las siguientes:

- $MediaEst = CantE \times CantE \times CoefM5$ es la media de trabajo esperada para el 5% de las iteraciones. El coeficiente $CoefM5$ es un valor obtenido estadísticamente en la estimación para el 5% de las vueltas para datos con distribución al azar (Figura 7.7). En particular, $CoefM5 = 0.13973009$.
- $DesvioEst = 3 \times (MediaEst \times CoefD5/100)$ es el desvio estimado para el 5% de las iteraciones. El coeficiente $CoefD5$ es un valor obtenido estadísticamente en la estimación para el 5% de las vueltas para datos con distribución al azar. En particular, $CoefD5 = 0.25$ (se multiplica por 3 para saber si pertenece a una distribución Normal).
- $PeorCasoEst = CantE \times CantE \times CoefP5$ es la cantidad de trabajo al 5% de las vueltas en el peor caso. El coeficiente $CoefP5$ es un valor analítico para el caso de secuencias totalmente invertidas[2]. En particular, $CoefP5 = 0.1445$.
- $DifEst = PeorCasoEst - MediaEst$ es la diferencia de estimación (usado en cálculos posteriores).

La estimación del trabajo total que debe realizar cada tarea, basado en lo que hizo en el 5% de las iteraciones, es:

for $i = 1$ to $k$
  if $((TrabajoP[i] - MediaEst) > desvioEst)$
    then
      $PorcAux = (TrabajoP[i] - MediaEst) \times 100/DifEst$
      $TrabajoT[i] = TrabajoP[i] \times 100/(CoefPT5 - (PorcAux \times CoefDN5/100))$
    else
      $TrabajoT[i] = TrabajoP[i] \times 100/CoefPT5$

donde $CoefPT5$ es el valor obtenido en la estimación del porcentaje del trabajo total que se realiza en el 5% de las iteraciones para datos con distribución al azar. En particular, $CoefPT5 = 11.378$. $CoefDN5$ es el estimado de la diferencia entre el peor caso y el normal. En particular, $CoefDN5 = 1.678$.

[2] $CoefP5 = \frac{(CantE^2 - 95\%CantE^2) \times 2.96}{CantE^2}$. El valor 2.96 corresponde a $CantCo + Coef \times CantInt$

Cuando $((TrabajoP[i] - MediaEst) > desvioEst)$, el trabajo realizado se encuentra más alejado del estimado que lo esperado para el caso de datos aleatorios. Por este motivo, se realiza una interpolación con el peor caso tomando un porcentaje de la diferencia. En caso contrario, se lo considera dentro del caso normal.

*Distribuir los datos reservados inicialmente*

Sea $max$ la tarea que se estima que realizará más trabajo ($TrabajoT[max]$).

Se calcula el total de valores necesarios para que todos los procesos igualen en trabajo a la tarea $max$.

Diferencias con el maximo
$SumaI = 0$
for $i = 1$ to $k$
    $Diferencias[i] = TrabajoT[max] - TrabajoT[i]$
    $CantidadExtra[i] = sqrt(Diferencias[i]/CoefRTT)$
    $SumaI = SumaI + CantidadExtra[i]$

Para obtener $CantidadExtra[i]$ se utiliza el coeficiente $CoefRTT$, que representa la relación entre el trabajo y la cantidad de elementos. En particular, $CoefRTT = 1.2284$[3].

Si la cantidad total de datos necesarios para que todas equiparen a $max$ ($SumaI$) es mayor a lo que resta repartir ($Resto$), a cada proceso se le envía un segundo bloque de acuerdo a lo calculado anteriormente (siempre que queden valores).

$SumaI = 0$
for $i = 1$ to $k$
    if $(SumaI < Resto)$
      then
        $CantidadExtra[i] = Minimo(CantidadExtra[i], Resto - SumaI)$
        $SumaI = SumaI + CantidadExtra[i]$
      else
        $CantidadExtra[i] = 0$

Si no es así, se establece una cota máxima para el trabajo y se obtienen las diferencias de la misma con lo estimado para cada proceso. El valor $Suma$ contiene el total de esas diferencias.

[3]Está dado por 100/83/1.48

Finalmente se obtiene el porcentaje que representa cada diferencia con respecto al total, y se calcula el porcentaje de valores (tamaño de bloque) que debe enviarse a cada proceso. Para obtener una distribución adecuada, el valor de la cota se calcula en función del trabajo.

$Cota = 1.8 \times CantE \times CantE$
$Suma = 0$
for $i = 1$ to $k$
    $TrabajoE[i] = Cota - TrabajoT[i]$
    $Suma = Suma + TrabajoE[i]$
$SumaT = 0$
for $i = 1$ to $k$
    $TrabajoE[i] = sqrt(TrabajoE[i] \times 100/Suma)$
    $SumaT = SumaT + TrabajoE[i]$
$CantExt = 0$
for $i = 1$ to $k$
    $CantidadExtra[i] = ((TrabajoE[i] \times 100/SumaT) \times Resto)/100$
    $CantExt = CantExt + CantidadExtra[i]$
$CantidadExtra[i] = Resto - CantExt$

En el siguiente Capítulo se presentan los resultados obtenidos utilizando el algoritmo *SPBCD*.

# Capítulo 8

# Resultados obtenidos

Entre los resultados que se obtuvieron, pueden analizarse el efecto del algoritmo de Sorting Paralelo con Balance de Carga Dinámico (*SPBCD*) sobre el trabajo realizado por las tareas y la distribución de carga del sistema. En particular, se lo compara con el Sorting Paralelo Sin Redistribución (*SPSR*) y el Sorting Paralelo Con Distribución Fija (*SBCDF*).

Las pruebas realizadas incluyeron diferentes tamaños de secuencias a ordenar (50000, 100000, 500000 y 1000000 de datos) y distinta cantidad de tareas utilizadas (4, 8 y 16). Sólo se muestran los resultados para ordenar secuencias de 1 millón de datos.

Se experimentó el método de sorting paralelo con balance de carga dinámico (*SPBCD*) variando el tamaño de *bloque2* (porción que se deja inicialmente sin repartir). Se presentan los datos obtenidos dejando como *bloque2* el 10, 20, 30 y 40% del tamaño de la secuencia original (*B10*, *B20*, *B30* y *B40* respectivamente) .

Otro de los parámetros que se variaron en las pruebas fue el tipo de secuencias utilizadas. Los resultados que se presentan se refieren a tres casos: (a) uno de los bloques enviados a las tareas (*bloque1*) totalmente invertido (*SUI*), (b) datos totalmente al azar (*STA*), y (c) la mitad de los bloques iniciales totalmente invertidos (*SMI*).

## 8.1 Trabajo Máximo

El tiempo de ejecución de un algoritmo paralelo se ve influido por lo que demora la tarea más lenta; en este caso, el proceso que termina último es aquel que debe realizar mayor cantidad de trabajo. Puede analizarse el trabajo máximo realizado en los algoritmos *SPBCD* y *SPCDF* con diferentes tamaños de *bloque2* en las Figuras 8.1, 8.2 y 8.3.

La diferencia en el trabajo máximo entre los dos algoritmos es mayor en el caso de *SUI*, ya que una sola de las tareas tiene mayor carga en *SPCDF*. También puede observarse

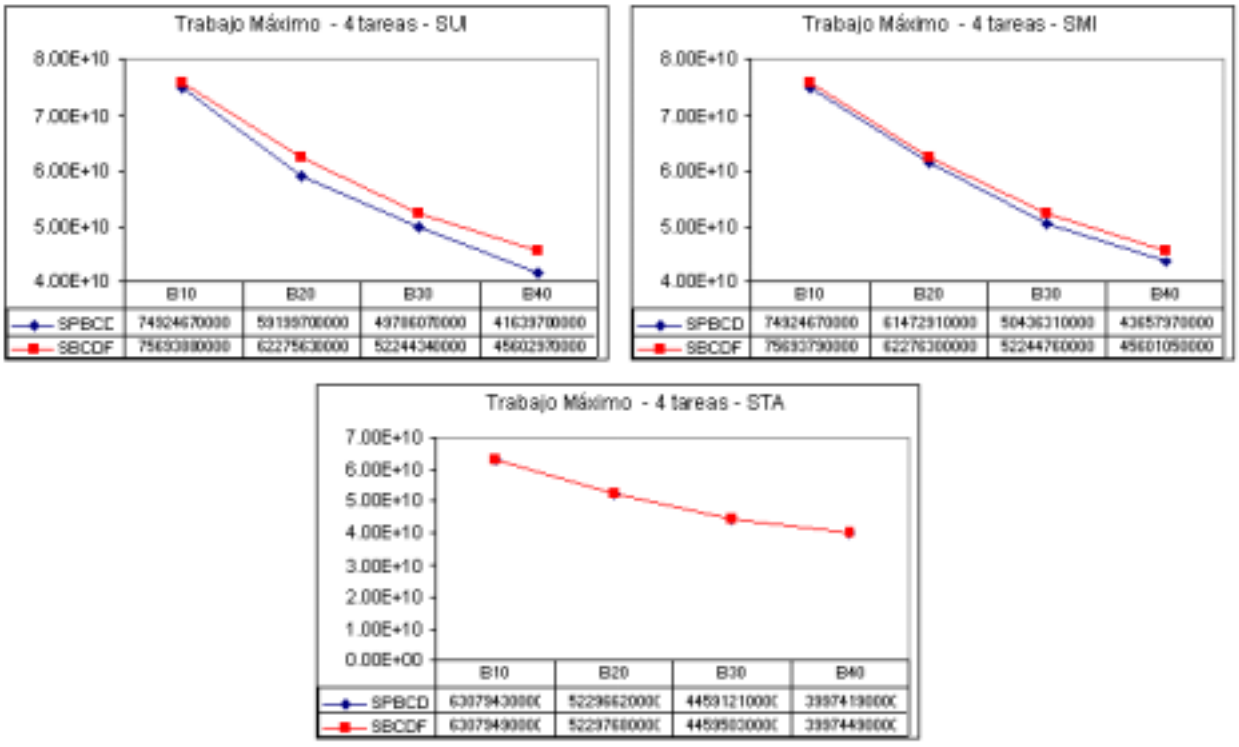


Figura 8.1: Trabajo Máximo para *SPBCD* y *SBCDF*, en secuencias *SUI*, *SMI* y *STA*, con 4 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

que la forma de las curvas se mantiene en general para las diferentes cantidades de tareas utilizadas.

## 8.2 Porcentaje de Reducción del Trabajo Máximo

la Figura 8.4 muestra la relación de reducción entre los trabajos máximos realizados en los métodos *SPBCD* y *SPCDF*. La reducción es calculada como $(TrabajoMaximoSPCDF - TrabajoMaximoSPBCD)/TrabajoMaximoSPCDF$, para la ejecución con 4, 8 y 16 tareas. Puede observarse que la reducción es mayor en el caso de *SUI*, en particular para tamaños de *bloque2* entre 30 y 40%. Para *STA*, la reducción es muy pequeña ya que inicialmente todas las secuencias son similares y por lo tanto la redistribución que puede realizarse es mínima.

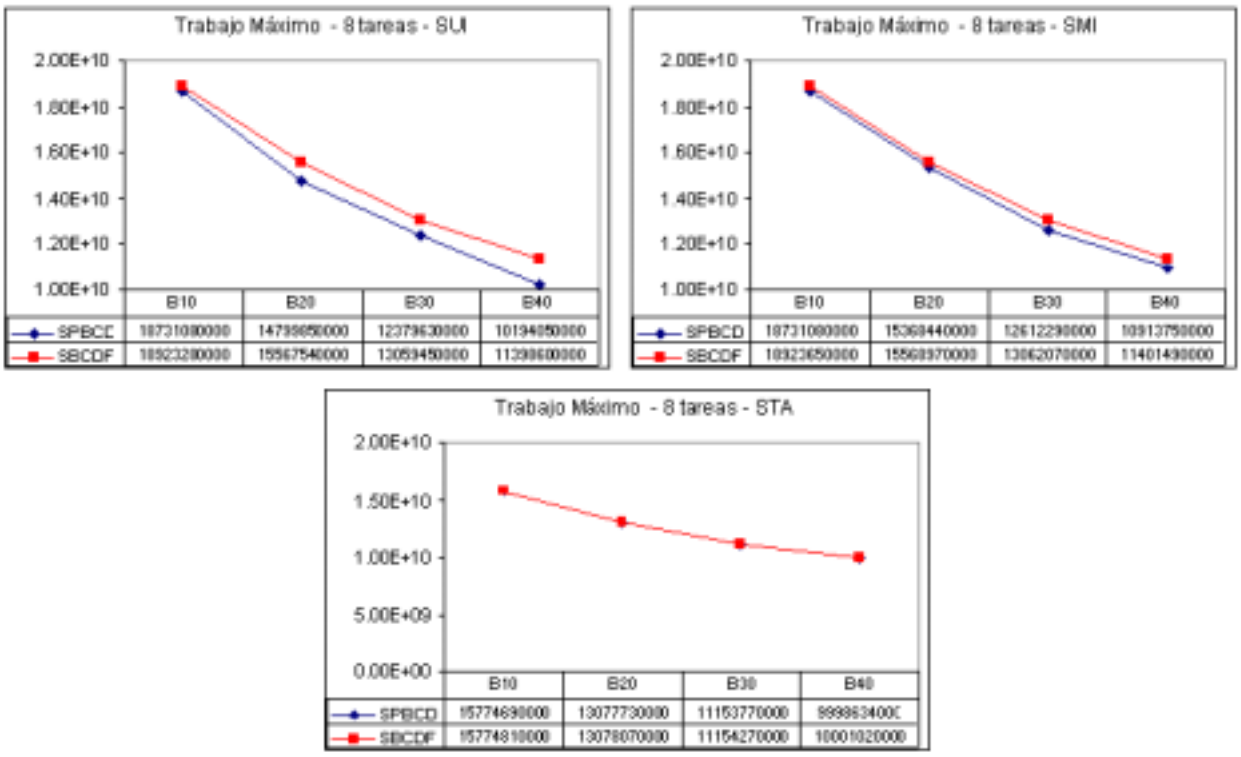


Figura 8.2: Trabajo Máximo para *SPBCD* y *SBCDF*, en secuencias *SUI*, *SMI* y *STA*, con 8 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

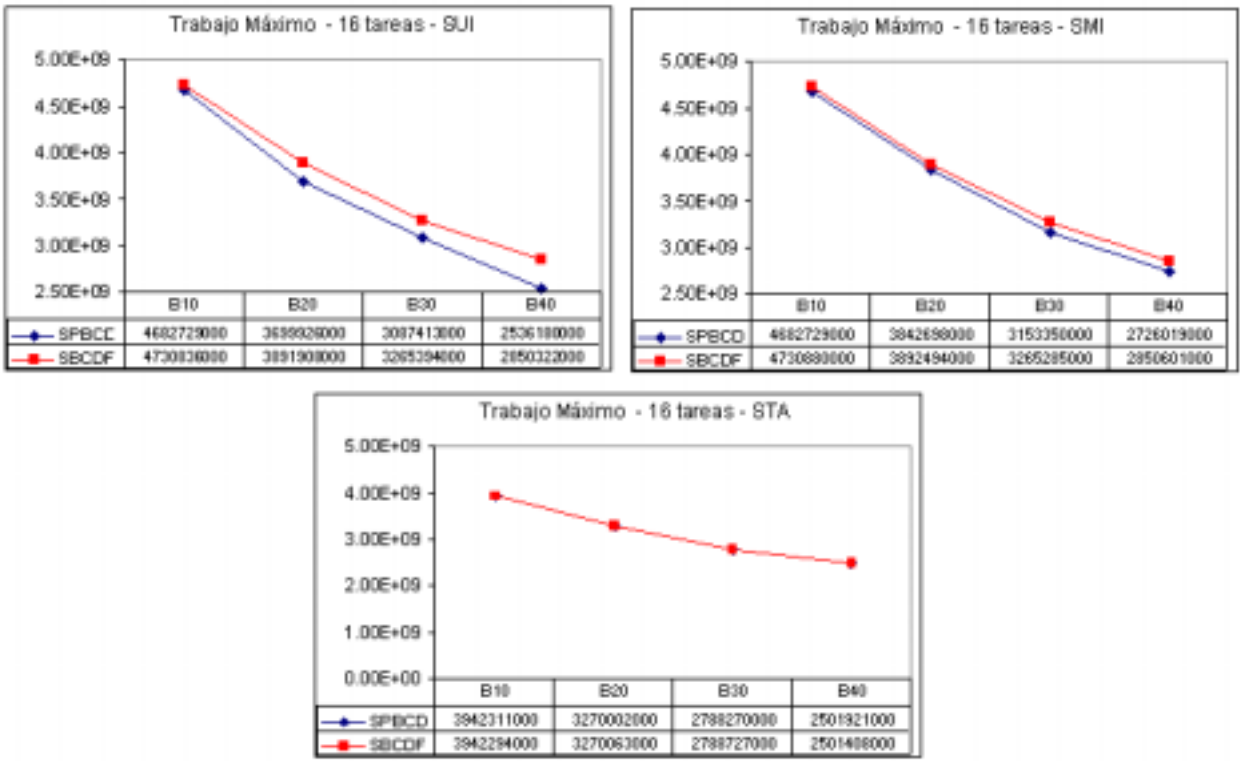


Figura 8.3: Trabajo Máximo para *SPBCD* y *SBCDF*, en secuencias *SUI*, *SMI* y *STA*, con 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40


Figura 8.4: Porcentaje de reducción del Trabajo Máximo, en secuencias *SUI*, *SMI* y *STA*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

## 8.3 Trabajo por tarea

Una forma de observar gráficamente el balance en la carga del sistema es a través del trabajo realizado por cada una de las tareas intervinientes en la resolución del problema. La Figura 8.5 muestra ésto (para 4 procesos) en los algoritmos *SPBCD* y *SPCDF*, y con secuencias de tipos *SUI*, *SMI* y *STA*. Puede observarse que para tamaños de *bloque2* entre 30 y 40% *SPBCD* logra que todas las tareas ejecuten prácticamente la misma cantidad de trabajo.

## 8.4 Balance de carga

Para analizar el balance de carga en el sistema se presentan la diferencia entre el trabajo máximo y mínimo realizado en *SPBCD*, *SPCDF* y *SPSR*, y qué porcentaje de desbalance representa esta diferencia respecto del promedio de trabajo ($(TrabajoMaximo - TrabajoMinimo)/TrabajoPromedio$). Las Figuras 8.6, 8.7 y 8.8 muestran esto para *SUI*, *SMI* y *STA*. Puede observarse que tanto la diferencia como el porcentaje se reducen en la mayoría de los casos para *SUI*, *SMI* y *STA*. Básicamente, los resultados menos satisfactorios se dan para *STA*, ya que en este caso en que las secuencias son totalmente aleatorias, la redistribución produce un efecto limitado. Las Figuras 8.9, 8.10 y 8.11 presentan los resultados para la diferencia entre el trabajo máximo y el trabajo promedio, y el porcentaje de desbalance como $(TrabajoMaximo - TrabajoPromedio)/TrabajoPromedio$.

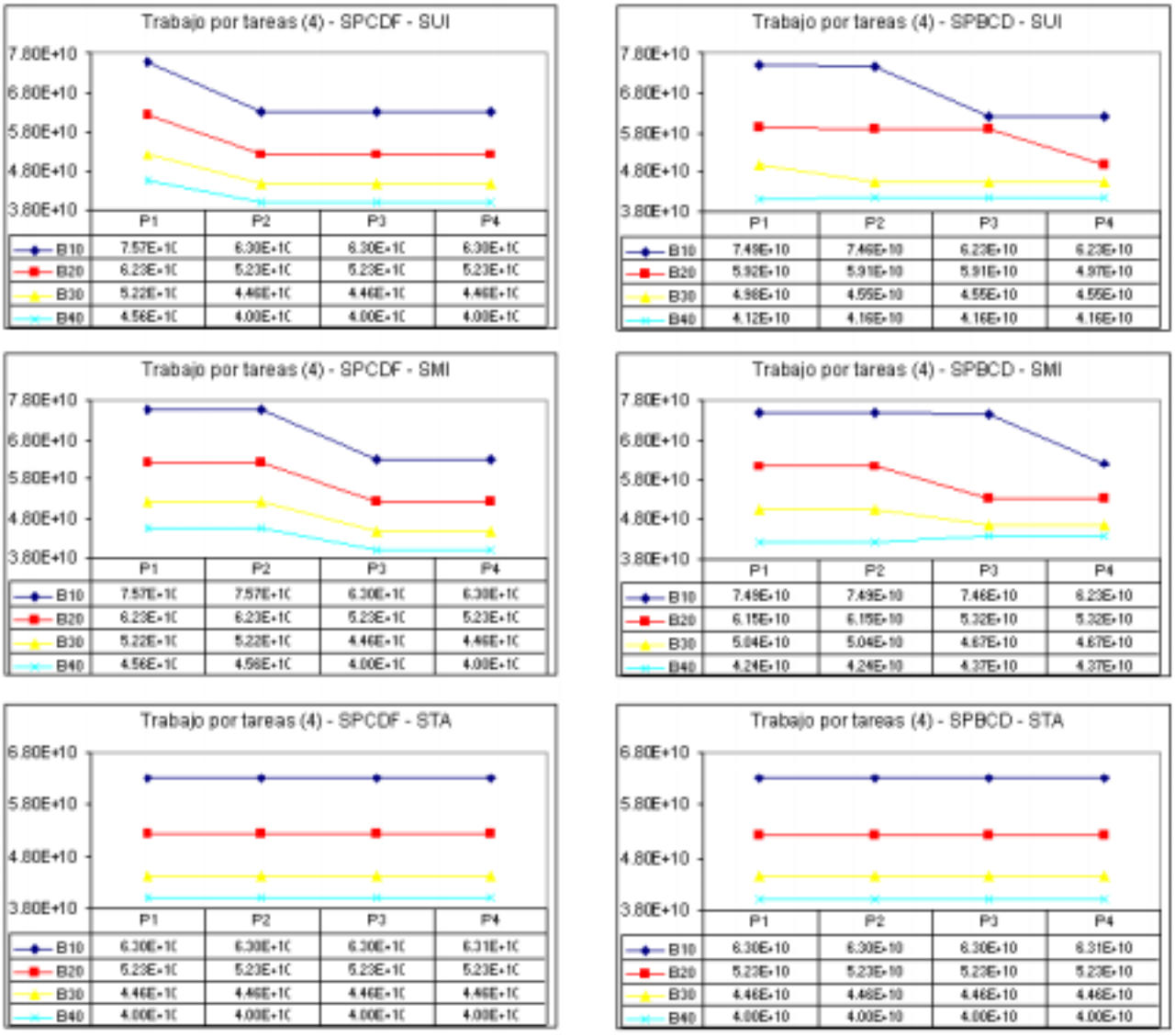


Figura 8.5: Trabajo por tareas, en secuencias *SUI*, *SMI* y *STA*, con 4 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

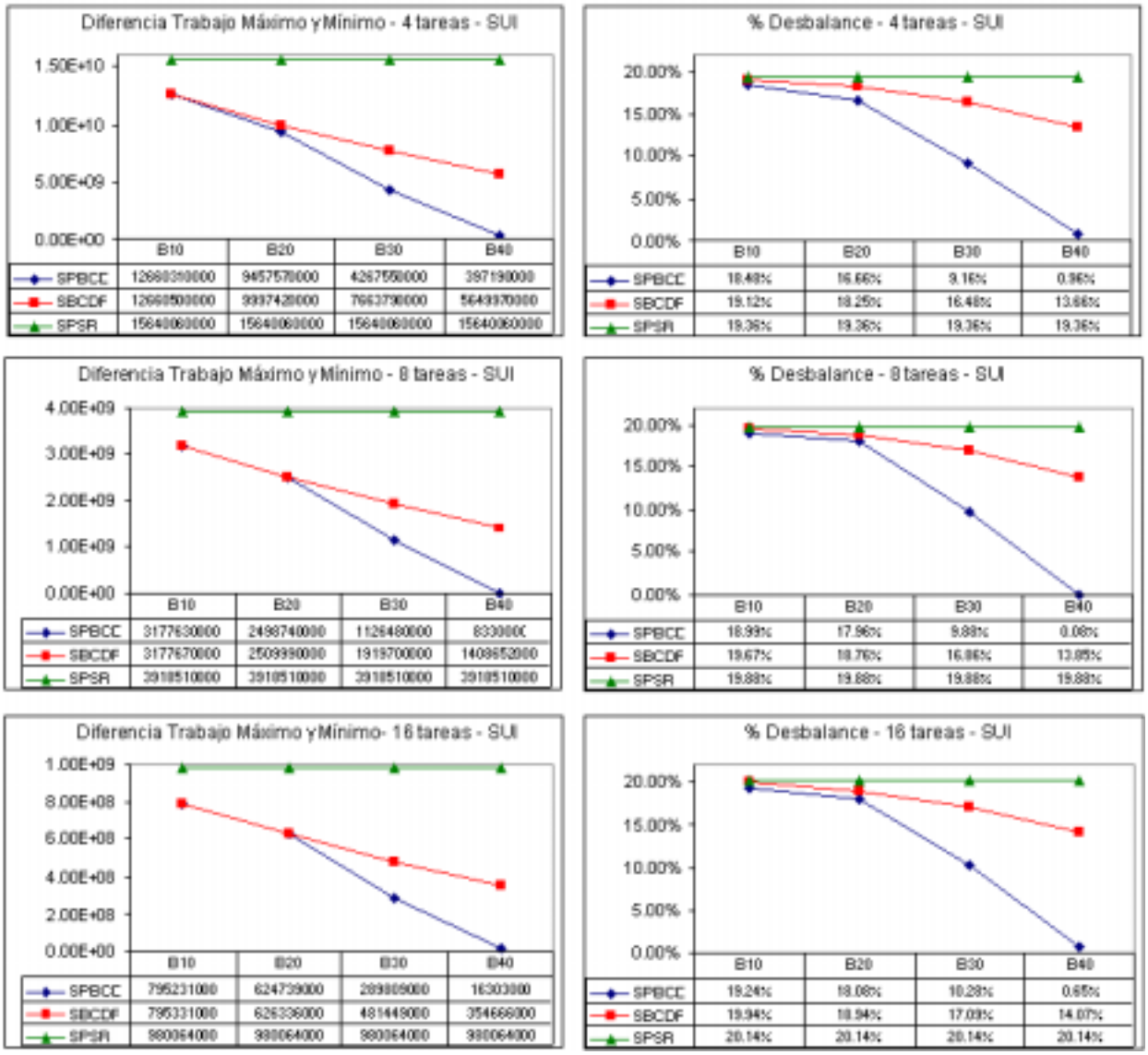


Figura 8.6: Diferencia entre Trabajo Máximo y Mínimo, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *SUI*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

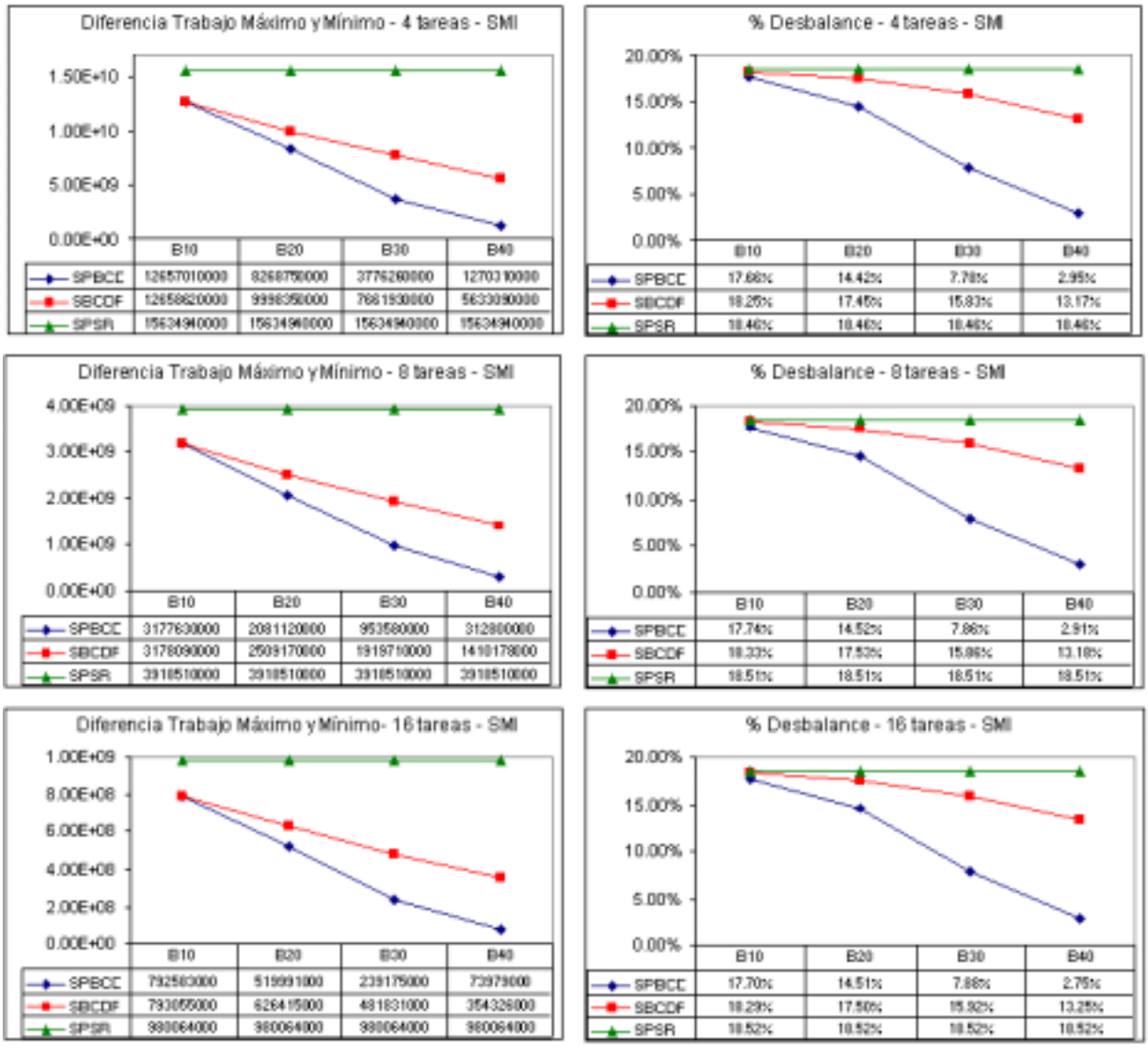


Figura 8.7: Diferencia entre Trabajo Máximo y Mínimo, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *SMI*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

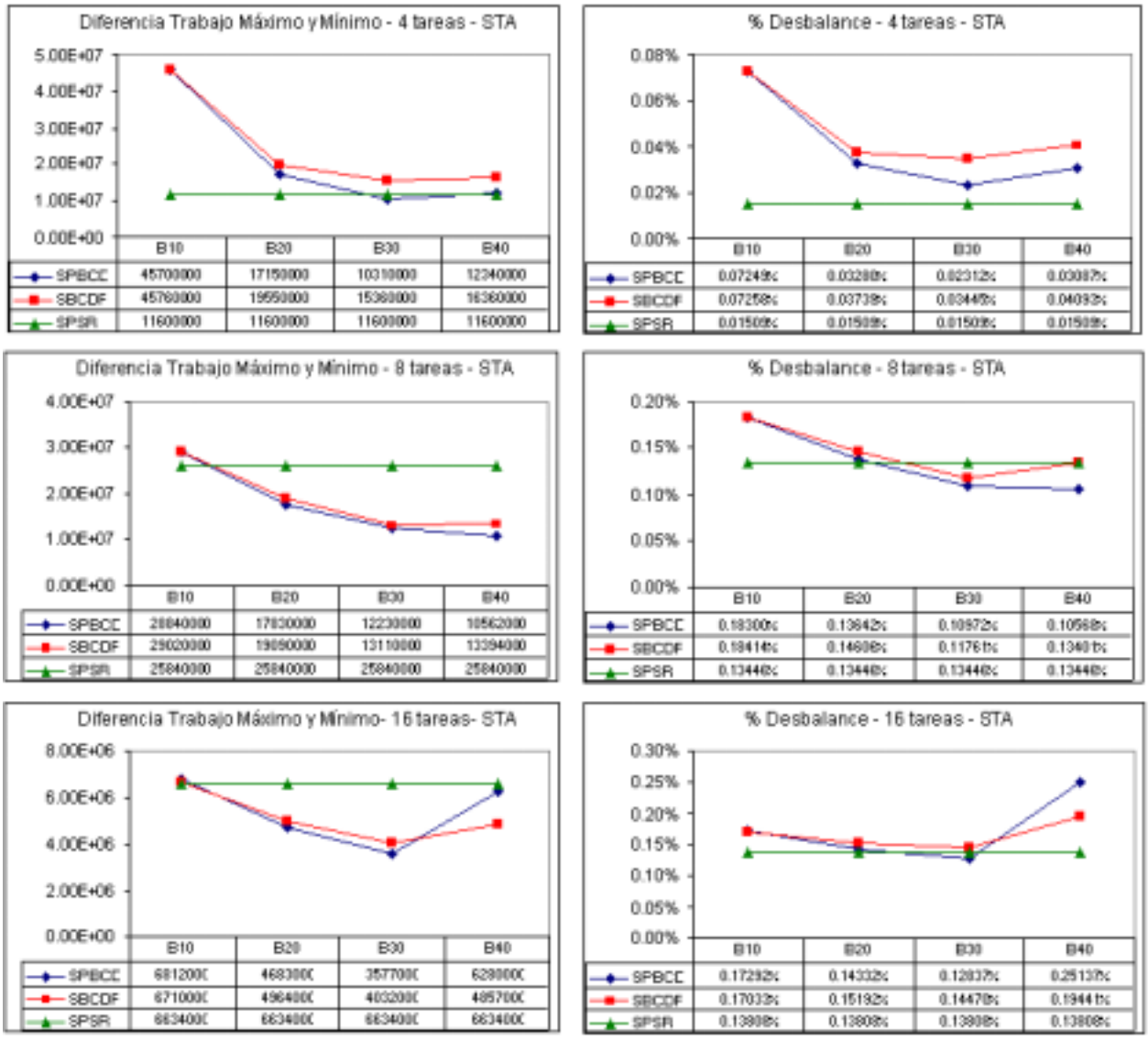


Figura 8.8: Diferencia entre Trabajo Máximo y Mínimo, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *STA*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

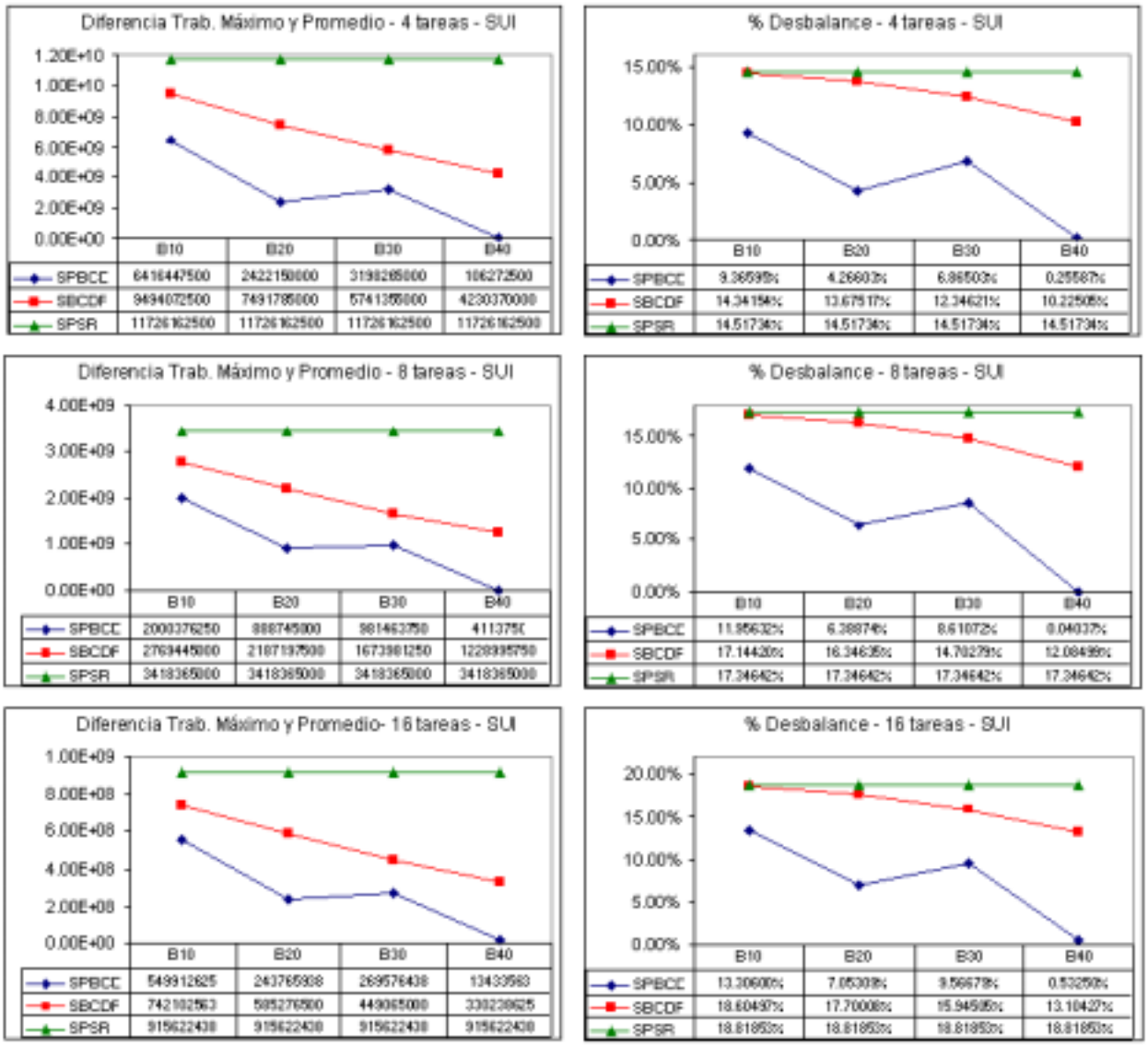


Figura 8.9: Diferencia entre Trabajo Máximo y Promedio, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *SUI*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

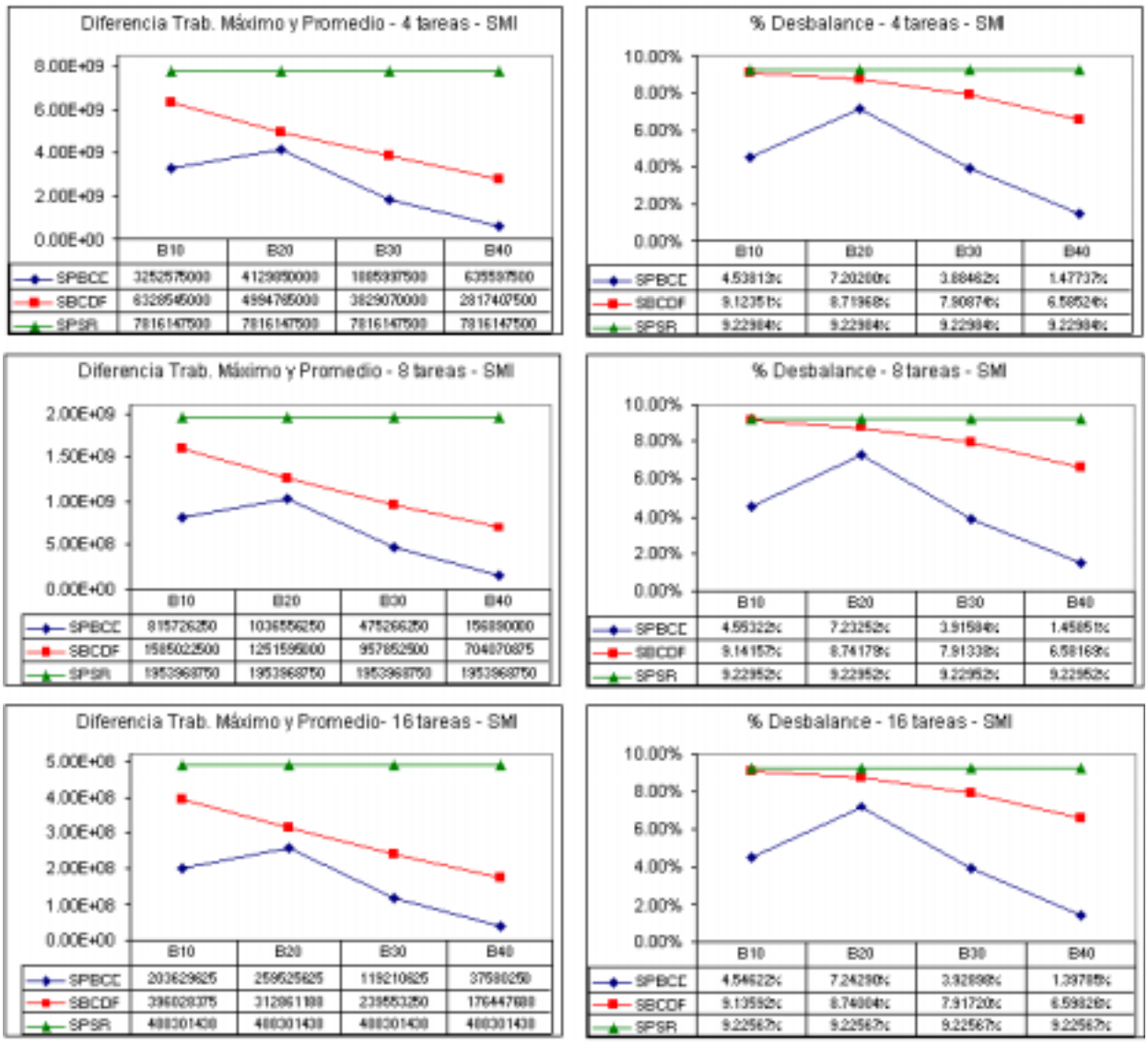


Figura 8.10: Diferencia entre Trabajo Máximo y Promedio, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *SMI*, con 4, 8 y 16 tareas y tamaño de $bloque2 = 10, 20, 30$ y 40


Figura 8.11: Diferencia entre Trabajo Máximo y Promedio, y Porcentaje de desbalance respecto del promedio de trabajo en secuencias *STA*, con 4, 8 y 16 tareas y tamaño de *bloque2*= 10, 20, 30 y 40

# Capítulo 9

# Conclusiones y Líneas Futuras

El tema de esta Tesis se inscribe en un área de interés actual dentro de los sistemas paralelos como es el balance de carga. Con el objetivo de brindar un marco conceptual, en la Parte I se realiza una revisión de aspectos teóricos generales de concurrencia y paralelismo, modelos, arquitecturas paralelas, tipos de paralelismo, paradigmas de computación paralela, y métricas de evaluación de performance.

Entre los objetivos del paralelismo se encuentran reducir el tiempo de ejecución y hacer uso eficiente de los recursos de cómputo. El balance de carga consiste en encontrar el mapeo de tareas (y los datos asociados) a una máquina paralela tal que cada procesador realice (aproximadamente) igual trabajo. Un algoritmo paralelo con un mapeo que balancea la carga de los procesadores incrementa la eficiencia global y, típicamente, reduce el tiempo de ejecución.

Sin embargo, el problema de asignación o mapeo es *NP*-completo para un sistema general con $n$ procesadores, y encontrar una asignación de costo mínimo es computacionalmente intratable salvo para sistemas muy chicos. Por este motivo pueden utilizarse enfoques alternativos como la relajación, el desarrollo de soluciones para casos particulares, o el uso de heurísticas que brindan soluciones subóptimas aceptables.

Existen una gran cantidad de técnicas de balance de carga tanto estáticas como dinámicas, y la aplicación de una u otra en gran parte depende del tipo de problema tratado. Si bien los métodos estáticos en general son más sencillos, no pueden utilizarse en aplicaciones donde la carga de trabajo (y el tiempo de cómputo asociado) no puede determinarse de antemano. Los métodos dinámicos pueden potencialmente mejorar la performance global redistribuyendo la carga durante la ejecución, aunque para ser efectivos no deben producir un gran overhead.

El *sorting* es una de las operaciones más comunes realizadas en una computadora, y su importancia está dada en que numerosas aplicaciones requieren que los datos se

encuentren ordenados para poder accederlos de manera más eficiente. Los problemas de sorting con gran volumen de datos por procesador son los más interesantes y sobre los cuales las máquinas paralelas actuales funcionan mejor, debido a su potencia de cómputo y capacidad de memoria.

Muchos de los métodos de balance de carga tratan con problemas en los que se cuenta con alguna manera de conocer cuál es la carga (por ejemplo, expresándola como una relación de la cantidad de puntos de entrada a procesar en una grilla). En una gran parte de los algoritmos de sorting, la carga de trabajo no depende del tamaño de los datos de entrada sino de una característica de los mismos como es el desorden inicial que presentan.

Esta Tesis presenta un nuevo método que balancea dinámicamente la carga basado en una distribución del trabajo utilizando un *estimador* para *predecir* la carga de trabajo pendiente. El método propuesto es una variante de Sorting by Merging Paralelo, una técnica basada en comparación; las ordenaciones en cada uno de los bloques en que se divide la secuencia a ordenar se realizan mediante Bubble Sort con centinela.

El *trabajo* a realizar, en términos de comparaciones e intercambios, se encuentra afectado por el *grado de desorden* de los datos. Se estableció la relación entre estos dos factores, estudiando además cómo influyen en el tiempo de ejecución del algoritmo. Se estudió la evolución de la cantidad de trabajo en diferentes porcentajes de iteraciones del algoritmo para distintos tipos de secuencias de entrada, observándose que el trabajo disminuye en cada iteración. Esto se utilizó para obtener una estimación del trabajo restante esperado a partir de una iteración determinada, y basarse en el mismo para corregir la distribución inicial de la carga.

Con esta idea, el método no distribuye inicialmente entre las tareas todos los datos a ordenar, sino que se reserva un porcentaje de ellos. Luego de una determinada cantidad de "vueltas" (en particular, con el 5% de las iteraciones), estima el trabajo restante de cada tarea basado en lo realizado, y distribuye dinámicamente los datos reservados de manera inversamente proporcional al trabajo restante estimado para cada una.

El esquema presentado utiliza el paradigma paralelo *master/slave*. La técnica de balance de carga que se propone es dinámica, iniciada por el emisor y basada en estimación. Se trata de un método sencillo y efectivo, que introduce un overhead muy limitado (ya que el cálculo del trabajo realizado y la estimación del restante no agrega excesivas operaciones) y poca comunicación (sólo $2\times$ la cantidad de tareas utilizadas).

Puede observarse en los resultados obtenidos que se balancea adecuadamente la carga de trabajo en un alto porcentaje de los casos, básicamente reduciendo el trabajo máximo a realizar en cada una de las tareas. La mejora en la distribución de la carga puede incrementarse cuando los datos a ordenar no son sólo un número sino una estructura más compleja, por lo que los intercambios que deben realizarse implican más trabajo. Dada la importancia del sorting dentro de la computación, la aplicación de esta técnica a distintos ámbitos es clara.

El método fue implementado sobre un cluster de PCs homogéneas (arquitectura paralela con comunicación por bus). Se realizó abundante experimentación con distintos tipos de secuencias y cantidad de tareas, y algunos de los resultados se muestran en el Capítulo 8.

Dentro de las líneas de trabajo futuras se encuentra la aplicación del mismo concepto de técnica de balance de carga basada en estimación a otros algoritmos de sorting y, de manera más general, a otros problemas de características similares.

Por otra parte, es posible extender el método propuesto a otras clases de arquitecturas paralelas. Entre ellas cabe mencionar a las máquinas de memoria compartida o de memoria compartida distribuida; el costo de adaptación en este caso es mínimo. También pueden tratarse los clusters heterogéneos, en cuyo caso deben tenerse en cuenta las diferentes velocidades de los procesadores tanto en el reparto inicial como en el cálculo de la redistribución del trabajo luego del 5% de las vueltas.

# Parte III

# Apéndices

# Apéndice A

# Complejidad de Funciones y Análisis de Orden

El análisis de orden y la complejidad asintótica de funciones son usadas extensamente para analizar la performance de algoritmos [209, 226].

## A.1 Complejidad de funciones

Tres tipos de funciones usadas para analizar algoritmos son:

1. **Funciones exponenciales**. Una función $f : \Re \to \Re$ es *exponencial* en $x$ si puede ser expresada en la forma $f(x) = a^x$ para $x$, $a \in \Re$. Por ejemplo, $2^x$, $3.4^{x+1}$, $3^{2x}$.

2. **Funciones polinomiales**. Una función $f : \Re \to \Re$ es *polinomial de grado* $b$ si puede ser expresada en la forma $f(x) = x^b$ para $x$, $b \in \Re$ y $b > 0$. Una función *lineal* es una función polinomial de grado 1, y una *cuadrática* una polinomial de grado 2. Por ejemplo, son funciones polinomiales $2x$, $3.5x^{3.2}$. Una función $f$ que es una suma de dos funciones polinomiales $g$ y $h$ es también una función polinomial cuyo grado es igual al máximo de los grados de $g$ y $h$.

3. **Funciones logarítmicas**. Una función $f : \Re \to \Re$ que puede ser expresada en la forma $f(x) = \log_b x$ para $b \in \Re$ y $b > 1$ es logarítmica en $x$. En la expresión, $b$ es la *base* del logaritmo. Por ejemplo, son logarítmicas $\log_{2.5} x$, $\log_2 x$.

Muchas funciones se pueden expresar como suma de dos o más. Una función $f$ se dice que *domina* a $g$ si $f(x)$ crece a una velocidad mayor que $g(x)$. Luego, $f$ domina a $g$ si y solo si $f(x)/g(x)$ es una función monótonamente creciente en $x$. En otras palabras, $f$

domina a $g$ si y solo si para cualquier constante $c > 0$, existe un valor $x_0$ tal que $f(x) > c.g(x)$ para $x > x_0$. Una función exponencial domina a una función polinomial y una polinomial a una logarítmica. La relación *domina* es transitiva, y así una exponencial domina a una logarítmica.

## A.2 Análisis de orden de funciones

En el análisis de algoritmos, con frecuencia es difícil o imposible derivar expresiones exactas para parámetros tales como tiempo de corrida, speedup o eficiencia. En muchos casos, una aproximación de la expresión exacta es adecuada. La aproximación puede ser ilustrativa del comportamiento de la función pues enfoca los factores críticos que influyen en el parámetro.

*La notación* $\Theta$. Dada una función $g(x)$, $f(x) = \Theta(g(x))$ si y solo si para constantes cualquiera $c_1$, $c_2 > 0$, existe $x_0 \geq 0$ tal que $c_1 g(x) \leq f(x) \leq c_2 g(x)$ para todo $x \geq x_0$.

*La notación* $O$. Dada una función $g(x)$, $f(x) = O(g(x))$ si y solo si para cualquier constante $c > 0$, existe $x_0 \geq 0$ tal que $f(x) \leq c\, g(x)$ para todo $x \geq x_0$. Se dice que $f(x)$ es *de orden a lo sumo* $g(x)$.

*La notación* $\Omega$. Dada una función $g(x)$, $f(x) = \Omega(g(x))$ si y solo si para cualquier constante $c > 0$, existe $x_0 \geq 0$ tal que $f(x) \geq c\, g(x)$ para todo $x \geq x_0$. Se dice que $f(x)$ es *de orden al menos* $g(x)$.

Ejemplo: Sean las funciones $D_1(t) = 1000\,t$, $D_2(t) = 100\,t + 20\,t^2$ y $D_3(t) = 25\,t^2$. Entonces, $D_3(t) = \Theta(D_2(t))$, $D_2(t) = \Theta(D_3(t))$, y además ambas son $\Theta(t^2)$. Por otra parte, $D_1(t) = O(D_2(t))$, $D_1(t) = O(t)$, $D_1(t) = O(t^2)$ y $D_2(t) = O(t^2)$. Finalmente, $D_3(t) = \Omega(D_1(t))$.

## A.3 Propiedades de funciones expresadas en notación de orden

Las notaciones de orden para expresiones tienen un número de propiedades útiles cuando se analiza la performance de algoritmos. Algunas de ellas son:

1. $x^a = O(x^b)$ si y solo si $a \leq b$.
2. $\log_a(x) = \Theta(\log_b(x))$ para todo $a$ y $b$.
3. $a^x = O(b^x)$ si y solo si $a \leq b$.

4. Para cualquier constante $c$, $c = O(1)$.
5. Si $f = O(g)$ entonces $f + g = O(g)$.
6. Si $f = \Theta(g)$ entonces $f + g = \Theta(g) = \Theta(f)$.
7. $f = O(g)$ si y solo si $g = \Omega(f)$.
8. $f = \Theta(g)$ si y solo si $f = \Omega(g)$ y $f = O(g)$.

# Apéndice B

# Modelo de Arquitectura y software

Para la experimentacin detallada en esta Tesis se utilizaron 17 PCs de un cluster. Los datos de las mquinas y la red de interconexin son los siguientes:

- PCs IBM con procesador Intel Pentium 4, 2392.099 Mhz, memoria RAM de 1 GigaByte, 8KBytes de cache L1, 256KBytes de cache L2,
- Red de interconexin: Ethernet 100 Mb, placas de red Intel EtherExpress Pro 100B, cableado con switch, ancho de banda 100 Mbits.

Los algoritmos fueron desarrollados en lenguaje C, utilizando la librera MPI (Message Passing Interface).

# Índice de Figuras

# Bibliografía


[1] Alok Aggarwal, Bowen Alpern, Ashok Chandra, and Marc Snir. A model for Hierarchical Memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[2] José Aguilar. An Approach to Mapping Parallel Programs on Hypercube Multiprocessors. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing. Funchal, Portugal. IEEE Press*, pages 221–224, February 1999.

[3] I. Ahmad, A. Ghafoor, and K. Mehrotra. Performance Prediction for Distributed Load Balancing on Multicomputer Systems. In *Proceedings of Supercomputing 1991*, pages 830–839, 1991.

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] S. Akl, M. Conrad, and A. Ferreira. Data-movement-intensive Problems: Two Folk Theorems in Parallel Computation Revisited. *Theoretical Computer Science*, pages 323–327, 1992.

[6] S. Akl and L. Lindon. Paradigms Admitting Superunitary Behaviour in Parallel Computation. Technical report, Queens University at Kingston, 1993.

[7] Selim G. Akl. *Parallel Computation. Models and Methods*. Prentice Hall, 1997.

[8] S.G. Akl, M. Cosnard, and A.G. Ferreira. Data-movement-intensive Problems: Two Folkore Theorems in Parallel Computation Revisited. *Theoretical Computer Science*, 95:323–337, 1992.

[9] M.A. Al-Mouhamed. Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs. *IEEE Transactions on Software Engineering*, 16(12):1390–1401, 1990.

[10] A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, 1995.

[11] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 12(2/3):72–109, 1994.

[12] Gail A. Alverson, William G. Griswold, David Notkin, and Lawrence Snyder. Abstractions for Portable, Scalable Parallel Programming. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):71–86, 1998.

[13] Nancy M. Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A Comparison of Parallel Sorting Algorithms on Different Architectures. Technical report no. 98/029, Department of Computer Science. Texas A&M University, 1996.

[14] Gene M. Amdahl. Validity of the Single-processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, pages 483–485, 1967.

[15] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Netwokrs of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[16] Thomas E. Anderson, David E. Culler, David A. Patterson, and NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[17] Gregory Andrews. *Concurrent Programming. Principles and Practice.* The Benjamin/Cummings Publising Company, 1991.

[18] Gregory Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming.* Addison Wesley, 2000.

[19] F.D. Anger, J.J. Hwang, and Y.C. Chow. Scheduling with Sufficient Loosely Coupled Processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[20] I.G. Angus, G.C. Fox, J.S. Kum, and D.W. Walker. *Solving Problems on Concurrent Processors, vol 2.* Englewood Cliffs, NJ.: Prentice-Hall, 1990.

[21] R.K. Arora and S.P. Rana. Heuristic Methods for Process Assignment in Distributed Computing Systems. *Information Processing Letters*, 11:199–203, December 1980.

[22] K.P. Arvind, Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Technical Report 114a, University of California, Irvine, December 1978.

[23] M. J. Atallah, editor. *Handbook of Algorithms and Theory of Computation*, chapter 46 - Algorithmic Techniques for Netwoks of Processors. CRC Press, 1999.

[24] D. Bader, B. Moret, and P. Sanders. *Experimental Algorithmics From Algorithm Design to Robust and Efficient Software, Vol. 2547 of Lecture Notes of Computer Science*, chapter Algorithm Engineering for Parallel Computation, pages 1–23. Springer Verlag, 2002.

[25] David A. Bader and Joseph JáJá. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. *Journal of Parallel and Distributed Computing*, 35(2):173–190, 1996.

[26] R. Bagrodia, K. M. Chandy, and M. Dhagat. UC - A Set Based Language for Data-parallel Programming. *Journal of Parallel and Distributed Computing*, 28(2):186–201, 1995.

[27] Fabrizio Baiardi, Sarah Chiti, Paolo Mori, and Laura Ricci. Integrating Load Balancing and Locality in the Parallelization of Irregular Problems. *Future Generation Computer Systems, Elsevier Science B.V.*, 17:969–975, 2001.

[28] Henri E. Bal and Matthew Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, pages 74–84, July-September 1998.

[29] S.T. Barnard and H.D. Simon. Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice and Experience*, 6(2):101–117, April 1994.

[30] S.T. Barnard and H.D. Simon. A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco*, pages 627–632, February 1995.

[31] M. Barton and G. Withers. Computing Performance as a Function of the Speed, Quantity, and Cost of the Processors. In *Proceedings of Supercomputing 89*, pages 759–764, 1989.

[32] K.E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the AFIPS Spring Joint Computer Conference, vol. 32, AFIPS Press, Reston, VA*, pages 307–314, 1968.

[33] J. Baxter and J.H. Patel. The LAST algorithm: A Heuristics-based Static Task Allocation Algorithm. In *Proceedings of International Conference on Parallel Processing, volume II*, pages 217–222, 1989.

[34] Wolfgang Becker. Dynamic Load Balancing for Parallel Database Processing. Faculty report no. 1997/08, Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1997.

[35] S. Ben Hansen and H.E. Bal. Integrating Task and Data Parallelism Using Shared Objects. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 317–324. ACM Press, 1995.

[36] Beowulf-Project. Beowulf Project Home Page. www.beowulf.org.

[37] Marsha J. Berger and Shahid H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987.

[38] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[39] Stefan Bischof, Ralf Ebner, and Thomas Erlebach. Parallel Load Balancing for Problems with Good Bisectors. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1998.

[40] D. Bjorner and O.N. Oest (Editors). *Towards a Formal Description of Ada*. Springer Verlag, 1980.

[41] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK Users' guide. SIAM, 1997.

[42] G. E. Blelloch. Programming Parallell Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[43] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An Experimental Analysis of Parallel Sorting Algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[44] Christopher A. Bohn and Gary B. Lamont. Load Balancing for Heterogeneous Clusters of PCs. *Future Generation Computer Systems, Elsevier Science B.V.*, 18:389–400, 2002.

[45] J.B. Boillat. Load Balancing and Poisson Equation in a Graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[46] S.H. Bokhari. Dual Processor Scheduling with Dynamic Reassignment. *IEEE Transactions on Software Engineering*, 5:341–349, July 1979.

[47] S.H. Bokhari. A Shortest Tree Algorithm For Optimal Assignments Across Space and Time in a Distributed Processor Systems. *IEEE Transactions on Software Engineering*, 7:583–589, November 1981.

[48] S.H. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing. *IEEE Transactions on Computers*, 37(1):48–57, January 1988.

[49] S.W. Bollinger and S.F. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Transactions on Computers*, 40(3):325–333, 1991.

[50] Per Brinch Hansen. *The Architecture of Concurrent Processes*. Prentice Hall, 1977.

[51] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, 1978.

[52] Per Brinch Hansen. *Studies in Computational Science. Parallel Programming Paradigms*. Prentice Hall, 1995.

[53] Robert K. Brunner and Laxmikant V. Kalé. Adapting to Load on Workstation Clusters. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation. Annapolis, Maryland.*, 1999.

[54] J. Bruno, E.G. Coffman, and R. Sethi. Scheduling Independent Tasks to Reduce Mean Finishing Time. *Communications of the ACM*, 17(7):382–387, July 1974.

[55] Marian Bubak, Wlodzimierz Funika, and J. Moscinski. Performance Analysis of Parallel Applications under Message Passing Environments. In *Proceedings of the 2nd International Conference on Parallel Processing and Applied Mathematics, Zakopane, Poland*, pages 414–422. R. Wyrzykowski, H. Piech, J. Szopa editors, September 1997.

[56] T.N. Bui and B.R. Moon. Genetic Algorithm and Graph Partitioning. *IEEE Transactions on Computers*, 45:841–855, 1996.

[57] A. Burns and W. Wellings. *Concurrency in Ada*. Cambridge University Press 2nd edition, 1998.

[58] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. BU Computer Science Technical Report 2002.

[59] Luis Miguel Campos and Isaac Scherson. Rate of Change Load Balancing in Distributed and Parallel Systems. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico*, pages 701–707, 1999.

[60] E. Carmona and M. Rice. Modeling the Serial and Parallel Fractions of a Parallel Algorithm. *Journal of Parallel and Distributed Computing*, 13:286–298, 1991.

[61] Soumen Chakrabarti, James Demmel, and Katherine A. Yelick. Modeling the Benefits of Mixed Data and Task Parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 74–83, 1995. URL: citeseer.nj.nec.com/chakrabarti95modeling.html.

[62] Wai-Yip Chan and Chi-Kwong Li. A New Technique of List Scheduling Algorithm for Heterogeneous Processors Systems. In *Proceedings of International Conference on Computer Applications in Industry and Engineering*, pages 56–60, 1997.

[63] Wai-Yip Chan and Chi-Kwong Li. Scheduling Tasks in DAG to Heterogeneous Processor System. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, IEEE Press*, pages 27–31, January 1998.

[64] Barbara Chapman, Matthew Haines, Piyush Mehrotra, Hans Zima, and John Van Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(2), 1997.

[65] V. Chaudhary and J.K. Aggarwal. A generalized Scheme for Mapping Parallel Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[66] Po-Jen Chuang and Chih-Ming Wu. An Efficient Recognition-Complet Processor Allocation Strategy for k-ary n-cube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):485–490, May 2000.

[67] E.G. Coffman. *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.

[68] Michele Colajanni and Michele Cermele. DAME: An Environment for Preserving the Efficiency of Data-Parallel Computations on Distributed Systems. *IEEE Concurrency*, 5(1):41–55, January-March 1997.

[69] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[70] R. Cole. Parallel Merge Sort. *Siam Journal of Computing*, 17(4):770–785, 1988.

[71] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proccedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[72] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. On the Effectiveness of Different Diffusive Load Balancing Policies in Dynamic Applications. In *Proceedings of the Conference on High Performance Computing and Networking Europe '98 (HPCN '98), Amsterdam (NL), April 1998, Lecture Notes in Computer Science, No.1401, Springer-Verlag (D)*, pages 274–283, 1998.

[73] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive Load-Balancing Policies for Dynamic Applications. *IEEE Concurrency*, 7(1):22–31, January-March 1999.

[74] A. Cortés, M. Planas, A. Ripoll, M.A. Senar, and E. Luque. Applying Load Balancing in Data Parallel Applications Using DASUD. In *Proceedings of 10th European PVM/MPI Users Group Conference (EuroPVM/MPI03). Lecture Notes in Computer Science*, September 2003.

[75] A. Cortés, A. Ripoll, F. Cedó, M.A. Senar, and E. Luque. On the Convergence of SID and DASUD Load Balancing Algorithms. Technical report. pirdi-8/98, Universidad Autónoma de Barcelona, 1998.

[76] A. Cortés, A. Ripoll, F. Cedó, M.A. Senar, and E. Luque. An Asynchronous and Iterative Load Balancing Algorithm for Discrete Load Model. *Journal of Parallel and Distributed Computing. Academic Press.*, 62:1729–1746, 2002.

[77] A. Cortés, A. Ripoll, M.A. Senar, F. Cedó, and E. Luque. On the Stability of a Distributed Dynamic Load Balancing Algorithm. In *IEEE Proceedings of International Conference on Parallel and Distributed Systems (ICPADS.*, pages 435–446, December 1998.

[78] A. Cortés, A. Ripoll, M.A. Senar, and E. Luque. Performance Comparison of Dynamic Load Balancing Strategies for Distributed Computing. In *Proceedings of Thirty-Second Annual Hawaii Internationsl Conference on System Sciences. IEEE Press*, volume 8, January 1999.

[79] A. Cortés, A. Ripoll, M.A. Senar, P. Pons, and E. Luque. On the performance of Nearest-Neighbors Load Balancing Algorithms in Parallel Systems. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing. Funchal, Portugal. IEEE Press.*, December 1999.

[80] Cray. Cray Systems. www.cray.com.

[81] Mark E. Crovella and Thomas J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Supercomputing '94*. IEEE Computer Society, 1994.

[82] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Notices (USA)*, 28(7):1–12, 1993.

[83] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the Conference on Supercomputing*, pages 262–273. ACM, 1993.

[84] G. Cybenko. Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[85] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[86] Sivarama P. Dandamudi. Sensitivity Evaluation of Dynamic Load Sharing in Distributed Systems. *IEEE Concurrency*, 6(3):62–72, July-September 1998.

[87] DDM. Domain Decomposition. www.ddm.org.

[88] E. Dekel, D. Nassimi, and S. Sahni. Parallel Matrix and Graph Algorithms. *SIAM Computing*, 10(4):657–675, 1981.

[89] J.B. Dennis and K. Weng. An Abstract Implementation for Concurrent Computation with Streams. In *Proceedings of the International Conference on Parallel Processing*, pages 201–211, 1979.

[90] Frédéric Desprez and Frédéric Suter. Mixed Parallel Implementations of the Top Level of Strassen and Winograd Matrix Multiplication Algorithms. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

[91] M. Dhagat, R. Bagrodia, and M. Chandy. Integrating task and data parallelism in UC. In *Proceedings of the 1995 International Conference on Parallel Processing*, volume 2, pages 29–36, 1995.

[92] R. Diekmann, A. Frommer, and B. Monien. Efficient Schemes for Nearest Neighbor Load Balancing. *Parallel Computing*, 25:789–812, 1999.

[93] R. Diekmann, D. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. Technical report, Department of Mathematics and Computer Science, University of Paderborn, Germany, 1995.

[94] R. Diekmann, B. Monien, and R. Preis. Using Helpful Sets to Improve Graph Bisections. Technical report tr-rf-94-008, University of Paderborn, 1994.

[95] N.J. Dimopoulos and V.V. Dimakopoulos. Optimal and Sub-optimal Processor Allocation for Hypercycle-based Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):175–185, February 1995.

[96] Jack Dongarra and Tom Dunigan. Message-Passing Performance of Various Computers. Technical Report CS-95-299, Knoxville, TN 37996, USA, 1996.

[97] D.L. Eager, E.D. Lazowska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6:53–68, 1986.

[98] K. Efe. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *Computer*, 15:50–56, June 1982.

[99] H. El-Rewini and T.G.. Leweis. Scheduling Parallel Programs Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, June 1990.

[100] H. El-Rewini and T. Lewis. *Distributed and Parallel Computing*. Manning Publications, 1998.

[101] D. Eppstein and Z. Galil. Parallel Algorithmic Techniques for Combinatorial Computation. *Annual Reviews in Computer Science*, pages 233–283, 1988.

[102] F. Ercal, J. Ramanujam, and P. Sadayappan. Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[103] S. Esquivel, S. Ferrero, R. Gallard, C. Salto, H. Alfonso, and M. Schütz. Enhanced Evolutionary Algorithms for Single and Multiobjective Optimization in the Job Shop Scheduling Problem. *Knowledge-Based Systems*, 15:13–25, 2002.

[104] S. Esquivel, C. Gatica, and R. Gallard. Conventional and Multirecombinative Evolutionary Algorithms for the Parallel Task Scheduling Problem. In *Proceedings of Applications of Evolutionary Computing, EvoWorkshops 2001: EvoCOP, Evoflight, EvoIASP, EvoLearn and EvoSTIM, Como, Italy. Egbert J. W. Boers and Jens Gottlieb and Pier Luca Lanzi and Robert E. Smith and Stefano Cagnoni and Emma Hart and Günther R. Raidl and H. Tijink (eds). Lecture Notes in Computer Science Vol. 2037, Springer Verlag. ISBN 3-540-41920-9*, pages 223–232, 2001.

[105] S. Esquivel, C. Gatica, and R. Gallard. Performance of Evolutionary Approaches for Parallel Task Scheduling under Different Representations. In *Proceedings of Applications of Evolutionary Computing, EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN, Kinsale, Ireland. Stefano Cagnoni and Jens Gottlieb and Emma Hart and Martin Middendorf and Günther R. Raidl (eds). Lecture Notes in Computer Science Vol. 2279, Springer Verlag. ISBN 3-540-43432-1*, pages 41–50, 2002.

[106] D.J. Evans and W.U.N. Butt. Dynamic Load Balancing Using Task-Transfer Probabilities. *Parallel Computing*, 19(8):897–916, August 1993.

[107] Charbel Farhat and Michel Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *International Journal for Numerical Methods in Engineering*, 36:745–764, 1993.

[108] E.B. Fernadez and B. Busseli. Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules. *IEEE Transactions on Computers*, 22(8):745–751, August 1973.

[109] Ursula Fissgus, Thomas Rauber, and Gudula Runger. A Framework for Generating Task Parallel Programs. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Maryland, US*, February 1999.

[110] M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[111] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 2nd ACM Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[112] Ian Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[113] C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salman, and D.W. Walker. *Solving Problems on Concurrent Processors, Volume 1.* Englewood Cliffs, NJ.: Prentice-Hall, 1988.

[114] James A. Freeman and David M. Skapura. *Neural Networks. Algorithms, Applications, and Programming Techniques.* Addison Wesley, 1991.

[115] M. Furuichi, K. Taki, and N. Ichiyoshi. A Multi-level Load Balancing Scheme for On-parallel Exhaustive Search Programs on the Multi-psi. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 50–59, 1990.

[116] M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the Theory of N-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[117] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[118] M. Gen and R. Cheng. *Genetic Algorithm and Engineering Optimization.* John Wiley and Sons, 1997.

[119] A. George, J.R. Gilbert, and J.W.H. Liu (ed). *Graph Theory and Sparse Matrix Computations.* The IMA Volumes in Mathematics and its Applications, No. 56, Springer Verlag, 1993.

[120] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling DAG on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

[121] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Trasactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[122] Alexandros V. Gerbessiotis and Constantinos J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27(6):809–822, 2001.

[123] B. Ghosh, S. Muthukrishnan, and M.H. Schultz. First and Second Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing. In *Proceedings of SPAA*, pages 72–81, 1996.

[124] P. Gibbons. A More Practical PRAM Model. In *Proccedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[125] M.D. Gil Montoya, C. Gil, and I. García. Load Balancing for a Class of Irregular and Dynamic Problems: Region Growing Image Segmentation Algorithms. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing. Funchal, Portugal. IEEE Press.*, February 1999.

[126] Ran Giladi and Niv Ahituv. SPEC as a Performance Evaluation Measure. *IEEE Computer*, 28(8):33–42, August 1995.

[127] J.R. Gilbert, G.L. Miller, and S.H. Teng. Geometric Mesh Partitioning: Implementation and Experiments. In *Proceedings of the Ninth International Parallel Processing Symposium, Santa Barbara, California*, pages 418–427, 1995.

[128] J.R. Gilbert and E. Zmijewski. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. *International Journal of Parallel Programming*, 16(6):427–449, 1987.

[129] David E. Goldberg. *Genetic Algorithms. In search, Optimization and Machine Learning*. Addison Wesley, 1998.

[130] O. Goldschmidt and D.S. Hochbaum. Polynomial Algorithm for the k-Cut Problem. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 444–451, 1988.

[131] Herman H. Goldstine. *The Computer*. Princeton University Press, 1972.

[132] J.A. Gonzalez, C. Leon, M.F. Piccoli, M. Printista, J.L. Roda, C.. Rodriguez, and F. Sande. The Collective Computing Model. *Journal of Computer Science and Technology*, 1(3):32–42, 2000.

[133] M.J. Gonzalez. Deterministic Processor Scheduling. *ACM Computing Surveys*, 9(3):173–204, September 1977.

[134] Michael T. Goodrich. Communication-Efficient Parallel Sorting. *Siam Journal on Computing*, 29(2):416–432, 2000.

[135] S. Gorlatch and C. Lengauer. Decomposition for Parallel Scan and Reduction. In *Proccedings of the Massively Parallel Programming Models*, pages 23–32, 1997.

[136] K.K. Goswami, M. Devarakonda, and R.K. Iyer. Prediction-Based Dynamics Load-Sharing Heuristics. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):638–648, June 1993.

[137] A. Gottlieb et al. The NYU Ultracomputer - Designing a MIMD, Shared Memory Parallel Computer. *IEEE Transactions on Computers*, pages 175–189, February 1983.

[138] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, (5):287–326, 1979.

[139] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *An Introduction to Parallel Computing. Design and Analysis of Algorithms*. Pearson Addison Wesley, 2nd Edition, 2003.

[140] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[141] Anshul Gupta and Vipin Kumar. Performance Properties of Large Scale Parallel Systems. *Journal of Parallel and Distributed Computing*, 19(3):234–244, 1993.

[142] John R. Gurd, Chris C. Kirkham, and Ian Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, 1985.

[143] John Gustafson. Forgotten Aspects of Parallel Performance. Slides of talk at 6th International IEEE Symposium on High Performance Distributed Computing Techniques and Applications, Las Vegas, Nevada. July. 1997. http://www.scl.ameslab.gov/Publications/publicationsjohn.html.

[144] John Gustafson. The Program of Grand Challenge Problems: Expectations and Results. Slides of talk at 2nd International Symposium on Parallel Algorithms/Architectures Synthesis, Fukushima, Japan. March 1997. http://www.scl.ameslab.gov/Publications/Japan97/Japan97.html.

[145] John Gustafson. Development of Parallel Methods For a 1024-Processor Hypercube. *Siam Journal on Scientific and Statistical Computing*, 9:532–553, 1988.

[146] John Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[147] John Gustafson. Fixed Time, Tiered Memory, and Superlinear Speedup. In *Proceedings of the Fifth Conference on Distributed Memory Computing (DMCC)*, 1990.

[148] John Gustafson. SLALOM: The Race Continues. *Supercomputing Review*, March 1991.

[149] John Gustafson. The Consequences of Fixed Time Performance Measurement. In *Proceedings of the 26th Hawaii International Conference on System Sciences, Kauai, Hawaii*, volume 3, January 1992.

[150] John Gustafson. A Paradigm For Grand Challenge Performance Evaluation. In *Proceedings of the Toward Teraflop Computing and New Grand Challenge Applications Mardi Gras 94 Conference, Baton Rouge, Lousiana*, February 1994.

[151] John Gustafson, D. Rover, S. Elbert, and M. Carter. SLALOM: The First Scalable Supercomputer Benchmark. *Parallelogram*, February 1991.

[152] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. SLALOM: Is Your Computer On The List? *Journal of Parallel and Distributed Computing*, pages 52–59, July 1991.

[153] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. SLALOM: Surviving Adolescence. *Supercomputing Review*, December 1991.

[154] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. The Design of a Scalable, Fixed-time Computer Benchmark. *Journal of Parallel and Distributed Computing*, 11:338–401, 1991.

[155] John Gustafson and Quinn Snell. HINT: A New Way To Measure Computer Performance. In *Proceedings of the HICSS-28 Conference, Wailela, Maui, Hawai*, pages 392–401, January 1995.

[156] John Gustafson and Rajat Todi. Conventional Benchmarks as a Sample of the Performance Spectrum. *The Journal of Supercomputing*, 13:321–342, 1998.

[157] Habermann and Perry. *Ada for Experienced Programmers.* Addison Wesley, 1983.

[158] Harry E. Jordan Harry F. Jordan, Gita Alaghband. *Fundamentals of Parallel Computing.* Prentice Hall, 2002.

[159] David R. Helman, David A. Bader, and Joseph JáJá. A Randomized Parallel Sorting Algorithm with an Experimental Study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[160] D. Helmbold and C. McDowell. Modeling Speedup($n$) Greater Than $n$. In *Proceedings 1989 International Conference on Parallel Processing*, volume 3, pages 219–225, 1989.

[161] Bruce Hendrickson and Robert Leland. Multidimensional Spectral Load Balancing. Technical Report SAND93-0074, Sandia National Laboratories. Albuquerque - NM, January 1993.

[162] Bruce Hendrickson and Robert Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal Scientific Computing*, 16(2):452–469, 1995.

[163] Robert Hercht-Nielsen. *Neurocomputing.* Addison Wesley, 1991.

[164] T. Hey, A. Dunlop, and E. Hernandez. Realistic Parallel Performance Estimation. *Parallel Computing*, 23:5–21, 1997.

[165] High Performance Fortran Forum. High Performance Fortran Language specification. January 1997. dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/hpf-report.html.

[166] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11), 1993.

[167] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[168] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[169] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[170] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence* . MIT Press, 1975.

[171] J.-W. Hong, X.-N. Tan, and M. Chen. From Local to Global: An Analysis of Nearest Neighbor Balancing on Hypercube. In *Proceedings of ACM-SIGMETRICS*, pages 73–82, 1988.

[172] S.H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, 10:160–166, 1990.

[173] T.C. Hu. Parallel Sequencing and Assembly Line Problems. *Oper. Research*, 19(6):841–848, November 1961.

[174] Y.F. Hu and R.J. Blake. An Improved Diffusion Algorithm for Dynamic Load Balancing. *Parallel Computing*, 25:417–444, 1999.

[175] Hugues. Sorting Networks and the END Search Algorithm. http://www.cs.brandeis.edu/~hugues/sorting_networks.html.

[176] C.-C. Hui, M. Hamdi, and I. Ahmad. SPEED: A Parallel Platform for Solving and Predicting the Performance of PDEs on Distributed Systems. *Concurrency: Practice and Experience*, 9:537–568, 1996.

[177] Chi-Chung Hui and Samuel T. Chanson. Allocating Task Interaction Graph to Processors in Heterogeneous Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):908–925, September 1997.

[178] Chi-Chung Hui and Samuel T. Chanson. Hydrodynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 10(11):1118–1137, November 1999.

[179] Chi-Chung Hui and Samuel T. Chanson. Improved Strategies for Dynamic Load Balancing. *IEEE Concurrency*, pages 58–67, July-September 1999.

[180] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[181] K. Hwang and Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[182] Kai Hwang. *Advanced Computer Architecture. Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[183] IBM. IBM SP2. www.ibm.com.

[184] Inmos. *Occam*. Prentice Hall, 1990.

[185] Inmos. *Transputer*. Prentice Hall, 1990.

[186] A. Itzkowitz and A. Schuster. Distributed Shared Memory: Bridging the Granularity Gap. Position Paper. www.cs.technion.ac.il/Labs/Milipede.

[187] A. Itzkowitz, A. Schuster, and L. Shalev. The Milipede Virtual Parallel Machine for NT/PC Clusters. Position Paper. www.cs.technion.ac.il/Labs/Milipede.

[188] J.F. JáJá. *An introduction to parallel algorithms*. Addison Wesley, 1992.

[189] J.F. JáJá and K.W. Ryu. The Block Distributed Memory Model for Shared Memory Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium, Cancún, México*, pages 752–756, 1994.

[190] Jack Jean, Karen Tomko, Vikram Yavgal, Robert Cook, and Jignesh Shah. Dynamic Reconfiguration to Support Concurrent Applications. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 302–303, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[191] Minsoo Jeon and Dongseung Kim. Parallel Merge Sort With Load Balancing. *International Journal of Parallel Programming*, 31(1):21–33, June 2003.

[192] Muhammad Kafil and Ishfaq Ahmad. Optimal Task Assignment in Heterogeneous Distributed Computing Systems. *IEEE Concurrency*, 6(3):42–51, July-September 1998.

[193] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater Scalability for Parallel Molecular Dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[194] L.V. Kale. Comparing the Performance of Two Dynamic Load Distribution Methods. In *Proceedings of International Conference on Parallel Processing*, pages 8–12, 1988.

[195] Gregory Karagiorgos and Nikolaos M. Missirlis. Accelerated Diffusion Algorithms for Dynamic Load Balancing. *Information Processing Letters, Elsevier Science*, 84:61–67, 2002.

[196] A.H. Karp. Programming for Parallelism. *Computer*, 20(5):43–57, May 1987.

[197] G. Karypis and V. Kumar. *MeTiS - Unstructured Graph Partitioning and Sparse Matrix Ordering System*. University of Minnesota, 1995.

[198] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report TR 95-035, Dept. Computer Science, University of Minnesota, 1995.

[199] George Karypis and Vipin Kumar. Analysis of Multilevel Graph Partitioning. Technical Report TR 95-037, Dept. Computer Science, University of Minnesota, 1995.

[200] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–99, 1996.

[201] D.J. Kerbyson, E. Papaefstatuiou, J.S. Harper, S.C. Perry, and G.R. Nudd. Is Predictive Tracing too Late for HPC Users? In *Proceedings of High Performance Computing Conference 98*, pages 57–67. R.J. Allan, M.F. Guest, D.S. Henty, D. Nicole and A.D. Simpson (eds.), Plenum/Kluwer Publishing, 1999, 1998.

[202] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[203] D.E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching (2nd Edition)*. Addison-Wesley, 1998.

[204] P. T. Koch, J. S. Hansen, E. Cecchet, and X. Rousset de Pina. SciOS: An SCI Based Software Distributed Shared Memory. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, 1999. URL: www.cs.umd.edu/ keleher/wsdsm99.

[205] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[206] W.H. Kohler and K. Steiglitz. Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems. *Journal ACM*, 21(1):140–156, 1974.

[207] K. Kubota, K. Itakura, M. Sato, and T. Boku. Practical Simulation of Large-scale Parallel Programs and its Performance Analysis of the NAS Parallel Benchmarks. *Lecture Notes on Computer Science*, 1470:244–254, 1998.

[208] L. Kucera. Parallel Computation and Conflicts in Memory Access. *Information Processing Letters*, 14(2):93–96, 1982.

[209] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

[210] Vipin Kumar, Ananth Y. Grama, and Vempaty Nageshwara Rao. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[211] Vipin Kumar and Anshul Gupta. Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, 22:379–391, September 1994.

[212] Vipin Kumar, V. Nageshwara, and K. Ramesh. Parallel Depth First Search on the Ring Architecture. In *Proceedings of the 1988 International Conference on Parallel Processing, Penn. State University Press*, pages 128–132, 1988.

[213] Christian Kurmann, Felix Rauch, and Thomas M. Stricker. Cost/Performance Tradeoffs in Network Interconnects for Clusters of Commodity PCs. Technical report no. 391, Swiss Federal Institute of Technology Zurich, Institute for Computer Systems, January 2003.

[214] Uwe Kuster. Remarks for parallel algorithms and implementation. www.hlrs.de/organization/par/par_prog_ws/online/.

[215] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[216] S. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound. *Communications of the ACM*, 27(6):594–602, 1984.

[217] Zhiling Lan, Valerie E. Taylor, and Greg Bryan. A Novel Dynamic Load Balancing Scheme for Parallel Systems. *Journal of Parallel and Distributed Computing*, 62(12):1763–1781, December 2002.

[218] H.W. Lang. Sequential and Parallel Sorting Algorithms. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm.

[219] Laura Lanzarini, Cecilia Sanz, Marcelo Naiouf, and Fernando Romero. Mixed Alternative in the Assignment by Classes vs. Conventional Methods for Calculation of Individuals Lifetime in GAVaPS. In *Proceedings of 22nd International Conference on Information Technology Interfaces (ITI 2000), Pula, Croatia. June 2000.*, pages 383–389.

[220] E.L. Lawler and D.E. Wodd. Branch and Bound Methods: A Survey. *Operations Research*, 14:699–719, 1966.

[221] Cheol-Hoon Lee, D.Lee, and M. Kim. Optimal Task Assignment in Linear Array Networks. *IEEE Transactions on Computers*, 41(7):877–880, July 1992.

[222] Cheol-Hoon Lee and Kang G. Shin. Optimal Task Assignment in Homogeneous Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):119–129, February 1997.

[223] R.B. Lee. Empirical Results on the Speedup, Efficiency, Redundancy and Quality of Parallel Computations. In *Proceedings of the International Conference on Parallel Processing*, pages 91–96, 1980.

[224] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. Partitioned Parallel Radix Sort. *Journal of Parallel and Distributed Computing*, 62:656–668, September 2002.

[225] S.Y. Lee and J.K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computer*, 36(4):433–442, April 1987.

[226] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[227] Robert Leland and Bruce Hendrickson. An Empirical Study of Static Load Balancing Algorithms. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC 94)*, pages 682–685, 1994.

[228] Claudia Leopold. *Parallel and Distributed Computing. A survey of Models, Paradigms, and Approaches*. Wiley Series on Parallel and Distributed Computing. Albert Zomaya Series Editor, 2001.

[229] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Sun Microsystems Press (Prentice Hall), 1998.

[230] Ted G. Lewis and Hesam El-Rewini. *Introduction to Parallel Computing*, pages 30–33. Prentice Hall, Englewood Cliffs, 1992. The Gustafson-Baris Law.

[231] Jie Li and Hisao Kameda. Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems. *IEEE Transactions on Computers*, 47(3):322–332, March 1998.

[232] Keqin Li and Xian-He Sun. Average-Case Analysis of Isospeed Scalability of Parallel Computations on Multiprocessors. *International Journal of High Speed Computing*, 11(1):15–36, 2000.

[233] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19:1079–1103, 1993.

[234] Z. Li, P. Mills, and J. Reif. Models and Resources Metrics for Parallel and Distributed Computation. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 133–143, 1989.

[235] De-Ron Liang and Satish Tripathi. On Performance Prediction of Parallel Computations with Precedent Constraints. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):491–508, May 2000.

[236] Ching-Jung Liao and Yeh-Ching Chung. Tree-Based Parallel Load-Balancing Methods for Solution-Adaptive Finite Element Graphs on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 10(4):360–370, April 1999.

[237] F.C.H. Lin and R.M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.

[238] Kathy J. Liszka, John K. Antonio, and Howard Jay Siegel. Problems with Comparing Interconnection Networks: Is an Alligator Better Than an Armadillo? *IEEE Concurrency*, 5(4):18–28, 1997.

[239] V.M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 11:1384–1397, November 1988.

[240] Peter Kok Keong Loh, Wen Jing Hsu, Cai Wentong, and Nadarajah Sriskanthan. How Network Topology Affects Dynamic Load Balancing. *IEEE Parallel and Distributed Technology*, 4(3):25–35, Fall 1996.

[241] R. Luling, B. Monien, and F. Ramme. Load Balancing in Large Networks: a Comparative study. In *Proceedings of 3rd IEEE Symposium on parallel and distributed processing*, pages 686–689, 1991.

[242] Yu lung Lo and Yu chen Huang. Effective Skew Handling for Parallel Sorting in Multiprocessor Database Systems. In *Proceedings of the Ninth International Conference on Parallel and Distributed Systems (ICPADS'02). IEEE Press.*, 2002.

[243] Philip D. MacKenzie and Quentin F. Stout. Ultra-Fast Expected Time Parallel Algorithms. *Journal of Algorithms*, 26:1–33, 1998.

[244] N. Mansour. Allocating Data to Multicomputer Nodes by Physical Optimization Algorithms for Loosely Synchronous Computations. *Concurrency: Practice and Experience*, 4:557–574, 1992.

[245] N. Mansour and G.C. Fox. Allocating Data Distributed-Memory Multiprocessors by Genetic Algorithms. *Concurrency: Practice and Experience*, 6:485–504, 1994.

[246] Ignacio Martín and Francisco Tirado. Relationships Between Efficiency and Execution Time of Full Multigrid Methods on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):562–573, June 1997.

[247] O.C. Martin and S.W. Otto. Partitioning of Unstructured Meshes for Load Balancing. *Concurrency: Practice and Experience*, 7:303–314, 1995.

[248] D. McIlroy, P. McIlroy, and K. Bosti. Engineering Radix Sort. *Computing Systems*, 6(1), winter 1993.

[249] MGNet. Mgnet home page. www.mgnet.org.

[250] Zbigniew Michalewicz. *Genetic Algorithms+Data Structures = Evolution Programs. 3rd Edition*. Springer Verlag, 1996.

[251] M. Miller, C.D. Hansen, and C.R. Johnson. Simulated Steering with SCIRun in a Distributed Environment. In *Proceedings of the 4th International Workshop PARA 98*, pages 366–376. B. Kagstrom, J. Dongarra, E. Elmroth and J. Wasniewski (eds.) LNCS 1541, Springer, 1998.

[252] R. Miller and Q.F. Stout, editors. *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*. Cambridge, MA: MIT Press, 1996.

[253] M. Mitzenmacher, A. Richa, and R. Sitaraman. The Power of Two Random Choices: A Survey of the Techniques and Results. In Handbook of Randomized Computing, P. Pardalos, S. Rajasekaran, and J. Rolim, Eds. Kluwer, 2000.

[254] Burkhard Monien, Ralf Diekmann, Rainer Feldmann, Ralf Klasing, Reinhard Lling, Knut Menzel, Thomas Romke, and Ulf-Peter Schroeder. Efficient Use of Parallel and Distributed Systems: From Theory to Practice. *Computer Science Today. Recent Trends and Developments, J. van Leeuwen (ed.), Springer Verlag, Lecture Notes in Computer Science*, (1000):62–77, 1995.

[255] Burkhard Monien, Ralf Diekmann, and Reinhard Lling. Communication Throughput of Interconnection Networks. In *Proceedings 19th International Symposium on Mathematical Foundations of Computer Science. Lecture Notes in Computer Science No. 841. Springer Verlag*, pages 72–86, 1994.

[256] Anna Morajko, Eduardo César, Tomás Margalef, Joan Sorribes, and Emilio Luque. Dynamic Performance Tuning Environment. In *Proceedings of Euro-Par 2001. 7th International Euro-Par Conference. Manchester, UK*, pages 36–45. Springer Verlag, August 2001.

[257] D.E. Muller and F.P. Preparata. Bounds and Complexities of Networks for Sorting and for Switching. *Journal of the ACM*, 22(2):195–201, April 1975.

[258] Nasa. NAS Parallel Benchmarks. http://www.nas.nasa.gov.

[259] D. Nicol. Inflated Speedups in Parallel Simulations via malloc( ). *International Journal on Simulation*, 2:413–426, 1992.

[260] D.M. Nicol and D.R. O'Hallaron. Improved Algorithms for Mapping Pipelined and Parallel Computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[261] Kjell Nielsen. *Ada in Distributed Real-Time Systems.* McGraw-Hill, 1990.

[262] M. Nodine and J. Vitter. Large Scale Sorting in Parallel Machines. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, 1991.

[263] B.A. Norman and J.C. Bean. A Genetic Algorithm Methodology for Complex Scheduling Problems. *Nav. Res. Logist.*, 46(2):199–211, 1999.

[264] D. Nussbaum and A. Agarwal. Scalability of Parallel Machines. *Communications of the ACM*, 34(3):57–61, 1991.

[265] Scott Oaks and Henry Wong. *Java Threads, 2nd Edition.* OReilly, 1999.

[266] Eric W. Olsen and Stephen B. Whitehill. *Ada for Programmers.* Prentice Hall, 1983.

[267] R.H.J. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, 1989.

[268] Michael A. Palis, Jing-Chiou Liou, and David S.L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[269] Cherri M. Pancake. Is Parallelism for You? *IEEE Computational Science and Engineering*, 3(2):18–37, 1996.

[270] ParKBench. PARKBENCH (PARallel Kernels and BENCHmarks. http://netlib.org/parkbench/html.

[271] Srinivas Patil and Prithviraj Banerjee. A Parallel Branch and Bound Algorithm for Test Generation. *IEEE Transactions on Computer Aided Design*, 9(3), May 1990.

[272] G.F. Pfister. *In Search of Clusters*. Prentice Hall, 2nd Edition, 1998.

[273] Sundeep Prakash. *Performance Prediction of Parallel Programs*. PhD thesis, University of California, Los Angeles, 1997.

[274] J.F. Prins, S. Chatterjee, and M. Simons. Expressing Irregular Computations in Modern Fortran Dialects. In *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 1–6. Springer LNCS 1511, 1998.

[275] M. Printista. Modelos de Predicción en Computación Paralela. Master's thesis, Universidad Nacional del Sur - Argentina, 2001.

[276] W. Pugh. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande*, 1999. URL: www.cs.ucsb.edu/conferences/java99/program.html.

[277] X.-S. Qian and Q. Yang. Load Balancing on Generalized Hypercube and Mesh Multiprocessors with LAL. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 402–409, May 1991.

[278] M. Quinn. *Parallel Computing. Theory and Practice*. McGraw-Hill, 1994.

[279] Martin Raab and Angelika Steger. Balls into Bins - A Simple and Tight Analysis. *Lecture Notes in Computer Science*, 1518:159–165, 1998.

[280] Andrei Radulescu, Cristina Nicolescu, Arjan J.C. van Gemund, and Peter J. Jonker. CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of The 15th International Parallel and Distributed Symposium. San Francisco, California*, pages 132–139, 2001.

[281] Shankar Ramaswamy. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, University of Illinois, Urbana-Champaign, 1996.

[282] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

[283] Abhiram Ranade. Optimal Speedup for Backtrack Search on a Butterfly Network. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[284] S. Ranka and S. Sahni. *Hypercube Algorithms*. Springer-Verlag, 1990.

[285] S. Ranka, Y. Won, and S. Sahni. Programming a Hypercube Multicomputer. *IEEE Software*, 5:69–77, 1988.

[286] V. Rao and V. Kumar. On the Efficiency of Parallel Backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, 1993.

[287] T. Rauber and G. Runger. Scheduling of Data Parallel Modules for Scientific Computing. In *Proceedings of 9th SIAM Conference on Parallel Processing for Scientific Computing, SIAM (CD-ROM), San Antonio, USA*, 1999.

[288] C. Reeves. A Genetic Algorithm for Flow Shop Sequencing. *Computers and Operations Research*, 22:5–13, 1995.

[289] Alexander Reinefeld. Scalability of Massively Parallel Depth-First Search. *Parallel Processing of Discrete Optimization Problems. P.M. Pardalos, K.G. Ramakrishman and M.G.C. Resende, AMS Press, DIMACS Series.*, 22:305–322, 1995.

[290] C. Roig, A. Ripoll, M.A. Senar, F. Guirado, and E. Luque. Modelling Message-Passing Programs for Static Mapping. In *Euromicro Workshop on Parallel and Distributed Processing (PDP' 00). IEEE CS Press. USA*, pages 229–236, 1999.

[291] K.W. Ross and D.D. Yao. Optimal Load Balancing and Scheduling in a Distributed Computer System. *Journal of Association for Computing Machinery*, 38(3):676–690, 1991.

[292] Sartaj Sahni and Venkat Thanvantri. Performance Metrics: Keeping the Focus on Runtime. *IEEE Parallel and Distributed Technology*, 4(1):43–56, 1996.

[293] Vikram Saletore and L.V. Kale. Consistent Linear Speedup to a First Solution in Parallel State-Space Search. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 227–233, 1990.

[294] F. Sande. *The Collective Computing Model*. PhD thesis, Universidad de La Laguna, España, 1998.

[295] Peter Sanders and Thomas Hansch. Efficient Massively Parallel Quicksort. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 13–24, 1997.

[296] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, Mass: MIT Press, 1989.

[297] I. Schoinas, B. Falsafi, M.D. Hill, J.R. Larus, and D.A. Wood. Sirocco: Cost-effective Fine-grain Distributed Shared Memory. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1998. URL: www.computer.org/proceedings.

[298] A. Schoneveld, J. de Ronde, P. Sloot, and J. Kaandorp. A Parallel Cellular Genetic Algorithm Used in Finite Element Simulation. *Parallel Problem Solving from Nature*

*(PPSN IV), H.-M. Voigt, W. Ebeling, I. Rechenberg, H.-P. Schwefel (Eds.)*, pages 533–542, 1996.

[299] Arjen Schoneveld, Peter M.A. Sloot, Martin Lees, and Erwan Karyadi. A Framework for Dynamic Load Balancing: A Case Study on Explosive Containment Simulation. *Future Generation Computer Systems, Elsevier Science B.V.*, 26:737–751, 2000.

[300] V.J. Schuster. Parallel Fortran for HP systems. Slides of talk at Hewlett-Packard High-Performance Computing Users Group Meeting. Chicago. March. 1999. www.pgroup.com/presentations/hpcug99/index.htm.

[301] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1995.

[302] M.A. Senar, A. Ripoll, A. Cortes, and E. Luque. Clustering and Reassignment-base Mapping Strategy for Message-Passing Architectures. In *International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 98), IEEE CS Press, USA*, pages 415–421, 1998.

[303] R. Sethi. Scheduling Graphs on Two Processors. *SIAM Journal of Computing*, 5(1):73–82, March 1976.

[304] SGI. SGI Origin 2000. www.sgi.com.

[305] Hongzhang Shan and Jaswinder Pal Singh. Parallel Sorting on Cache-coherent DSM Multiprocessors. In *Proceedings of SC99 Conference on High Performance Networking and Computing*, 1999.

[306] D.L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[307] Hanmao Shi and Jonathan Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[308] Yuan Shi. Reevaluating Amdahl's Law and Gustafson Law. http://joda.cis.temple.edu/ shi/docs/amdahl/amdahl.htm.

[309] Yuan Shi and Wes Powers. Timing Models - A Parallel Program Performance Analysis and Experimentation Method. http://joda.cis.temple.edu/ shi/super98/npc/index.html.

[310] Y. Shih and J. Fier. Hypercube Systems and Key Applications. In *Parallel Processing for Supercomputers and Artificial Intelligence (K. Hwang and D. DeGroot Eds. McGraw-Hill*, pages 203–243, 1989.

[311] Wei Shu and L.V. Kale. A Dynamic Scheduling Strategy for the Chare-Kernel System. In *Proceedings of Supercomputing 89*, pages 389–398, 1989.

[312] Wei Shu and Min-You Wu. Runtime Incremental Parallel Scheduling (RIPS) on Distributed Memory Computers. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):637–649, June 1996.

[313] G.C. Sih and E.A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):75–187, February 1993.

[314] E. Silva and M. Gerla. Queueing Network Models for Load Balancing in Distributed Systems. *Journal of Parallel and Distributed Computing*, 12:24–38, 1991.

[315] H.D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.

[316] J. Simon and J.-M. Wierum. The Latency-of-Data-Access Model for Analysing Parallel Computation. *Information Processing Letters. Special Issue on Models of Computation*, 1998.

[317] Jens Simon and Jens-Michael Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Euro-Par 96*, volume 1124 of *LNCS*, France, Lion, August 27-29 1996. Springer.

[318] D. Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.

[319] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[320] Quinn O. Snell and John L. Gustafson. An Analytical Model of the HINT Performance Metric. In *Proceedings of the Supercomputing 96 Conference*. ACM Press and IEEE Computer Society Press, 1996.

[321] M. Snir. On Parallel Searching. *Siam Journal on Computing*, 14(3):688–708, 1985.

[322] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The complete Reference*. Cambridge, MA: MIT Press, 1996.

[323] Lawrence Snyder. Type Architecture. Shared Memory and the Corollary of Modest Potential. *Ann. Rev. Computer Science*, 1:289–318, 1986.

[324] S. Sofianopoulou. The Process Allocation Problem: A Survey of the Application of Graph-Theoretic and Integer Programming Approaches. *Journal of the Operational Research Society*, 43(5):407–413, 1992.

[325] E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear Speedup for Parallel Backtracking. *Lecture Notes in Computer Science, Springer Verlag*, 297:985–993, 1988.

[326] E. Speight and J.K. Bennett. Brazos: A third generation DSM system. In *Proceedings of the 1997 USENIX Windows/NT Workshop*, pages 95–106, 1997.

[327] W. R. Stevens. *Unix Network Programming.* Prentice Hall, 1990.

[328] H.S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, 3:85–93, January 1977.

[329] H.S. Stone. Critical Load Factors in Distributed Computer Systems. *IEEE Transactions on Software Engineering*, 4:254–258, May 1978.

[330] H.S. Stone and S.H. Bokhari. Control of Distributed Processes. *Computer*, 11:97–106, July 1978.

[331] J. Subhlok and B. Yang. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12. ACM Press, 1997.

[332] Xian-He Sun. The Relation of Scalability and Execution Time. In *Proceedings of the International Parallel Processing Symposium 96*, 1996.

[333] Xian-He Sun. Performance Range Comparison via Crossing Point Analysis. *Lecture Notes in Computer Science, Springer Verlag*, 1388, March 1998.

[334] Xian-He Sun. Scalability versus Execution Time in Scalable Systems. *Journal of Parallel and Distributed Computing*, (62):173–192, 2002.

[335] Xian-He Sun and John Gustafson. Toward a Better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

[336] Xian-He Sun and L. Ni. Another View on Parallel Speedup. In *Proceedings Supercomputing 90*, pages 324–333, 1990.

[337] Xian-He Sun and Lionel M. Ni. Scalable Problems and Memory-Bounded Speedup. *Journal of Parallel and Distributed Computing*, 19:27–37, September 1993.

[338] Xian-He Sun, Mario Pantano, Thomas Fahringer, and Zhaohua Zhan. SCALA: A Framework for Performance Evaluation of Scalable Computing. In *Proceedings of the 4th Workshop on High Level Parallel Programming Models and Supportive Environments (HIPS 99), Lecture Notes in Computer Science No. 1586, Springer Verlag*, pages 49–62, 1999.

[339] Xian-He Sun and Diane Thiede Rover. Scalability of Parallel Algorithm-Machine Combinations. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):599–613, June 1994.

[340] Xian-He Sun and J. Zhu. Shared Virtual Memory and Generalized Speedup. In *Proceedings 8th International Parallel Processing Symposium*, pages 637–643, 1994.

[341] Xian-He Sun and Jianping Zhu. Performance Considerations of Shared Virtual Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):1185–1194, November 1995.

[342] Xian-He Sun and Jianping Zhu. Performance Prediction: A Case Study Using a Scalable Shared-Virtual-Memory Machine. *IEEE Transactions on Parallel and Distributed Technology*, 4(4):36–49, 1996.

[343] Pierre Terdiman. Radix Sort Revisited. http://www.codercorner.com/RadixSortRevisited.htm.

[344] F. Tinetti and A. De Giusti. *Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos*. Editorial Exacta, 1998.

[345] Karen A. Tomko and Edward S. Davidson. Profile Driven Weighted Decomposition. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, May 1996.

[346] Karen Arnold Tomko. *Domain Decomposition, Irregular Applications, and Parallel Computers*. PhD thesis, University of Michigan, 1995.

[347] D.F. Towsley. Allocating Programs Containing Branches and Loops within a Multiple Processor System. *IEEE Transactions on Software Engineering*, 12:1018–1024, October 1986.

[348] S. Tucker Taft and R.A. Duff (Editors). *Ada 95 Reference Manual*. Springer LNCS 1246, 1997.

[349] J.C. Ueng, C.K. Shieh, and C.C. Lin. Design and Implementation of Proteus. In *Proceedings of 1st Workshop on Software Distributed Shared Memory*, 1999. URL: www.cs.umd.edu/ keleher/wsdsm99.

[350] L.G. Valiant. A Bridging Model for Parallel Computation. *Commnications of the ACM*, 33(8):103–111, 1990.

[351] D. Vanderstraeten and R. Keunings. Optimized Partitioning of Unstructured Finite-Element Meshes. *International Journal for Numerical Methods in Engineering*, 38:433–450, 1995.

[352] Alf Wachsmann and Rolf Wanka. Sorting on a Massively Parallel System Using a Library of Basic Primitives: Modeling and Experimental Results. In *European Conference on Parallel Processing*, pages 399–408, 1997.

[353] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Aspect Ratio. In *Proceedings VecPar'98, Porto, Portugal*, pages 381–394, 1998.

[354] W. Ware. The Ultimate Computer. *IEEE Spectrum*, 9:84–91, 1972.

[355] Jerrell Watts and Stephen Taylor. A Practical Approach to Dynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235–248, March 1998.

[356] E.A. West and A.S. Grimshaw. Braid: Integrating Task and Data Parallelism. In *Proceedings of Frontiers 95: 5th Symposium Frontiers of Massively Parallel Computations*, pages 211–219. IEEE CS Press, 1995.

[357] M.H. Willebeek-LeMair and A.P. Reeves. Local vs. Global Strategies for Dynamic Load Balancing. In *Proceedings of International Conference on Parallel Processing*, volume 1, pages 569–570, 1990.

[358] M.H. Willebeek-LeMair and A.P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[359] J.W.J. Williams. Algorithm 232. *Communications of the ACM*, 7(6):347–348, 1964.

[360] R.D. Williams. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. *Concurrency: Practice and Experience*, 3(5):457–481, October 1991.

[361] K. Windisch, V. Lo, and B. Bose. Contiguous and Non-Contiguous Processor Allocation Algorithms for K-ary n-cubes. In *Proceedings of 1995 International Conference on Parallel Processing*, pages II 164–168, August 1995.

[362] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of the Supercomputing 99 Conference on High Performance Networking and Computing, Portland, Oregon*, November 1999.

[363] J. Woo and S. Sahni. Hypercube Computing: Connected Components. *Journal of Supercomputing*, 3:209–234, 1989.

[364] J. Woo and S. Sahni. Computing Biconnected Components on a Hypercube. *Journal of Supercomputing*, 5(1):73–87, 1991.

[365] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[366] Patrick H. Worley. The Effect of Time Constraints on Scaled Speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990.

[367] Min-You Wu. On Parallelization of Static Scheduling Algorithms. *IEEE Transactions on Software Engineering*, 25(8):512–528, August 1997.

[368] Min-You Wu. On Runtime Parallel Scheduling for Processor Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):173–186, February 1997.

[369] Cheng-Zhong Xu and Francis C.M. Lau. Analysis of the Generalized Dimension Exchange Method for Dynamic Load Balancing. *Journal of Parallel and Distributed Computing*, 16(4):385–393, 1992.

[370] Cheng-Zhong Xu and Francis C.M. Lau. Decentralized Remapping of Data-parallel Computations with the Generalized Dimension Exchange Method. pages 414–421, 1994.

[371] Cheng-Zhong Xu and Francis C.M. Lau. Iterative Dynamic Load Balancing in Multicomputers. *Journal of Operational Research Society*, 45(7):786–796, July 1994.

[372] Cheng-Zhong Xu and Francis C.M. Lau. Optimal Parameters for Load Balancing with the Diffusion Method in Mesh Networks. *Parallel Processing Letters*, 4(2):139–147, 1994.

[373] Cheng-Zhong Xu and Francis C.M. Lau. The Generalized Dimension Exchange Method for Load Balancing in k-ary n-cubes and Variants. *Journal of Parallel and Distributed Computing*, 24(1):72–85, 1995.

[374] Cheng-Zhong Xu, Barkhard Monien, and Reinhard Luling. An Analytical Comparison of Nearest Neighbor Algorithms for Load Balancing in Parallel Computers. pages 472–479, 1995.

[375] Cheng-Zhong Xu, S. Tschoeke, and B. Monien. Performance Evaluation of Load Distribution Strategies in Parallel Branch-And-Bound Computations. Technical report, Department of Electrical and Computer Engineering, Wayne State University, 1995.

[376] T. Yamada and R.Ñakano. Genetic Algorithms for Jop Shop Scheduling Problems. In *Proceedings of the Modern Heuristic for Decision Support*, pages 67–81, 1997.

[377] Pamela Yang, Timothy M. Kunau, Bonnie Holte Bennett, Emmett Davis, and Bill Wren. Reconfigurable Parallel Sorting and Load Balancing on a Beowulf Cluster: HeteroSort. In *José D. P. Rolim (Ed.): Proceedings of 15 IPDPS Workshops, Parallel and Distributed Processing, Cancun, Mexico. Lecture Notes in Computer Science 1800 Springer Verlag. ISBN 3-540-67442-X*, pages 862–869, May 2000.

[378] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[379] Ron Zeno. A Reference of the Best-known Sorting Networks for up to 16 Inputs, May 11, 2002. http://home.san.rr.com/ronz/Articles/999SortingNetworksReferen.html.

[380] B.B. Zhou, X. Qu, and R.P. Brent. Effective Scheduling in a Mixed Parallel and Sequential Computing Environment. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing. Madrid, Spain. IEEE Press.*, pages 21–23, January 1998.

[381] Albert Y. Zomaya, Matthew Clemnts, and Stephan Olariu. A Framework for Reinforcement-Based Scheduling in Parallel Processor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):249–260, March 1998.